A New Automated Approach to Cloud Population

by

Hoki Tam

B.S., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2012

Certified by .....
Prof. Hari Balakrishnan
Professor
Thesis Supervisor

Certified by .....
Brad Meiseles
Director, R&D, VMware
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A New Automated Approach to Cloud Population

by

Hoki Tam

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents vCloud Populator, a new approach to automating the cloud population process on VMware vCloud Director. Cloud population is the process during which a cloud instance is populated with entities to reflect a particular desired state. It is generally done by developers manually to render their cloud into a certain state for a quick and easy manual testing of features. However, manual cloud population can be very complex and time-consuming. The main goal of the vCloud Populator project is to reduce the amount of time that developers spend on populating a vCloud Director instance manually. The vCloud Populator approach involves the design of a new domain-specific language to describe a desired cloud state and the implementation of a system to generate a dynamic execution plan to populate a vCloud Director instance given by the state captured by a input script written in the said language.

Thesis Supervisor: Prof. Hari Balakrishnan
Title: Professor

Thesis Supervisor: Brad Meiseles
Title: Director, R&D, VMware

Acknowledgments

This M.Eng thesis project is sponsored jointly by MIT and VMware via the MIT VI-A program.

I would like to begin by thanking everyone I know at VMware for always being helpful and answering any questions I have about both the VMware vCloud Director product and any other technical questions I come across. I would like to thank in particular my mentor Phil McGachey, for providing me with ample guidance throughout my internship at VMware. This project would not have been possible without his support and patience. I would also like to thank my supervisors, Joel Feldman and Brad Meiseles, for making this project possible and for the abundant guidance that they have provided me with.

I would also like to thank Professor Hari Balakrishnan, my faculty advisor at MIT, for taking the time to supervise this thesis despite his busy schedule. I would also like to thank everyone else I know at MIT for the support they have given me through the past four and a half years.

Finally, I must thank the VI-A Program for giving me such a wonderful opportunity to produce a thesis project in an industry setting. I have learned a lot through this valuable experience, and that would not have been possible without the VI-A Program.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Chapter 1

Introduction

This thesis describes the vCloud Populator, a project designed to automate the cloud population process for the VMware vCloud Director product [1]. VMware vCloud Director, built on top of the VMware vSphere [2] virtualization platform, provides Infrastructure-as-a-Service (IaaS) cloud computing to end users by introducing numerous abstractions designed to enhance the cloud experience for both the service provider and the cloud consumer. For example, a major abstraction that was introduced in vCloud Director is the organization. An organization is a group of users, and all user must belong to an organization. A user's action only affects his organization, and each organization's resources and data are generally private. The organization abstraction allows the service provider to host many different customer groups in the same cloud with an isolation mechanism in place to ensure security and privacy. Abstractions such as organizations and users are called **cloud entities**. The combination of cloud entities and their configuration on vCloud Director specifies a **cloud state**, which can be though of as a snapshot of a cloud at a point in time. The vCloud Populator project seeks to provide easy means for users to quickly populate a vCloud Director instance to reflect a desired cloud state using programmatic means.

1.1 Motivation

Every developer of vCloud Director has his own installation of the latest version of the product for testing purposes. Developers generally aim to keep their own cloud installation active with numerous cloud entities, as many functionalities must be tested with a populated cloud. For example, a developer cannot test the user creation function if an organization is not present, as all users must belong to an organization. However, because the product is under development, database schema designs are in an evolving state. Thus whenever new design changes are introduced to the database, the database must be cleaned to maintain compatibility with the changes. This removes all user-created cloud entities in the process. Developers must then repopulate their installation with new entities. Currently, this process is generally done manually by each developer through the web portal, consuming valuable development time and company resources.

1.2 Existing Approaches

There has been some work at VMware to automate the cloud population process. Firstly, developers can simply communicate programmatically with their installed cloud instance using the VMware vCloud API [3]. The VMware vCloud API is a RESTful [4] interface to the cloud, allowing for changes to be made to the cloud via a programmatic interface. To facilitate the programming, VMware has also presented software development kits (SDKs) in different languages, such as Java and PHP. Developers can thus create customized cloud population programs in the language of their choice, using the provided SDKs. This, however, may not be an ideal solution, since the SDKs follows an imperative model, requiring the developers to list out, step-by-step, the specific operations to execute in populating a cloud instance to match a particular desired state. This causes the coding process to be lengthy, and the resulting programs are generally quite verbose.

A recent project in the company populates the cloud by reading in a series of XML

configuration files describing a desired state of the cloud and calling the appropriate operations on the installed cloud instance via the Java SDK. This approach generates a dynamic execution plan to create cloud entities from a declarative input file. Using this approach, the developer only needs to describe the output state of the cloud, and can leave it to the project backend to determine the appropriate operations to perform on the cloud to match the desired output state. However, XML also tends to be quite verbose in general. Furthermore, the XML approach can become very repetitive in instances where the developer is only concerned about the general cloud structure rather than specific configurations for entities. For example, a developer may want to create hundreds of the same cloud entity without providing specific details for each of them. In XML, developers must repeatedly write out the correct XML tag along with other configurations that are necessary, whereas if they were using the SDK, a mere for loop can simplify matters.

1.3 vCloud Populator

Seeking to build and improve upon both the direct vCloud API/SDK approach and the XML approach, we implemented the vCloud Populator project. The vCloud Populator introduces a domain-specific language for cloud state specification. The domain-specific language, named **vCSL** (short for vCloud Specification Language), is designed to be considerably less verbose than both the vCloud API/SDK approach and the XML approach. vCSL, similar to the XML approach, is declarative, but provides the necessary programmatic controls that are available in the vCloud API/SDK approach as well.

The major contributions of the vCloud Populator project include four main components. The first is the vCloud Specification Language (vCSL for short), which is the domain-specific language that is designed to define and specify a desired state on vCloud Director. Secondly is the vCSL parser, which compiles any scripts written in vCSL into a cloud entity model graph, which is an in-memory representation of the desired vCloud Director state. This cloud entity model graph is the third major con-

tribution of this project. Finally, the vCloud Populator project includes an execution engine which traverses through the resulting cloud entity model graph and constructs the corresponding state on a vCloud Director instance.

1.4 Thesis Overview

The remainder of this thesis document describes the vCloud Populator project in detail. Chapter 2 outlines background information on the technology that the vCloud Populator project is based on. Chapter 3 presents the vCSL language in depth. Chapter 4 discusses the implementation of the different components in the vCloud Populator. Chapter 5 compares the manual population process with the vCloud Populator approach in some sample scenarios. Finally, Chapter 6 concludes this thesis and explores possible future enhancements to the vCloud Populator project.

Chapter 2

Background

This chapter gives brief background information on technology we use in implementing this project.

2.1 VMware vCloud Director

VMware vCloud Director is a cloud computing management software based on the Infrastructure-as-a-Service (IaaS) cloud computing model [5]. In this model, a **service provider** offers infrastructure resources such as computing resources, storage resources, and networking resources to their customers. Customers can pay for the actual amount of resources that they have used instead of needing to pay for the cost of purchasing and maintaining an entire data center. Given these infrastructure resources, the customers are then responsible for allocating and creating their own virtual machines and managing any applications that are running in these machines.

vCloud Director provides a web graphical user interface for customers to log in to the service provider's cloud and provision virtual machines on-demand, without assistance from the service provider. The service provider can then bill their customers based on how many virtual machines the customers had run, the amount of resources that these virtual machines have consumed, etc. vCloud Director also provides a REST [4] API, known as the vCloud API [3], allowing users to communicate programmatically with the vCloud Director server.

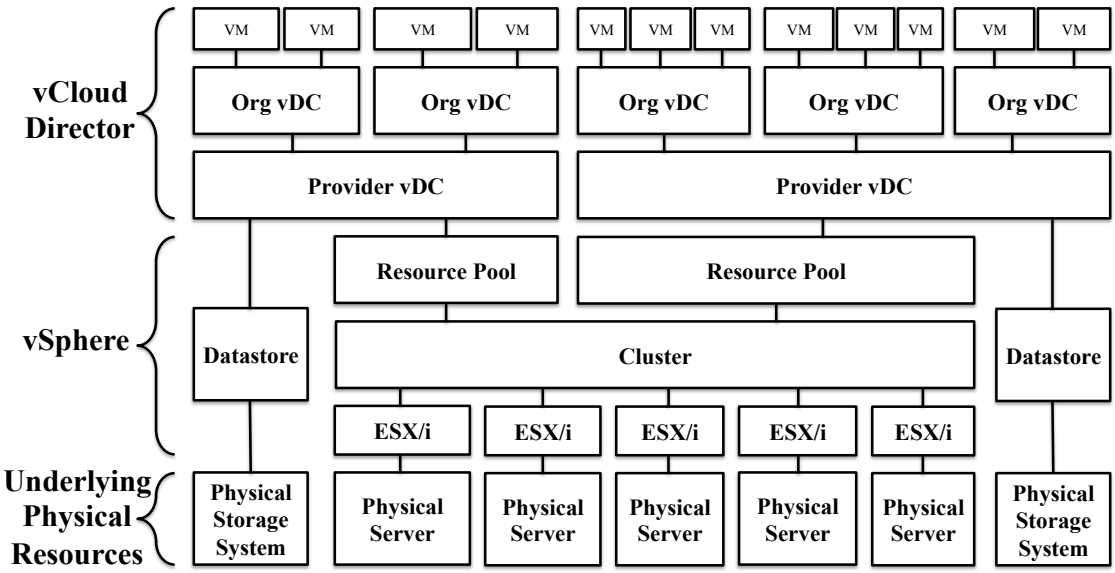

Figure 2-1: This diagram shows the stack of component layers between physical computing and storage resources and vCloud Director.

VMware's vCloud Director is built upon VMware vSphere. VMware vSphere manages all physical computing, storage, and networking resources. Physical hardware provides the actual, usable resources to serve all computing, storage, and networking needs. Physical x86 servers provides computing resources (CPU and memory), disk arrays provides storage resources (disk spaces), and physical network provides networking resources.

2.1.1 vSphere and vCloud Director

This section focuses mainly on the mechanics of how vSphere and vCloud Director works together to present physical resources to end users in a cloud environment. This discussion is limited to computing and storage resources only, since the vCloud Populator does not support networking in its current state. The stack of abstraction layers in Figure 2-1 shows how computing and storage resources are distributed from physical servers to virtual machines in vCloud Director.

The physical servers used in the vCloud stack run the VMware ESX or ESXi hypervisor, one component of vSphere. VMware's ESX/i hypervisor is a bare-metal hypervisor that virtualizes the computing resources provided by physical servers by

managing all available resources and presenting them to all virtual machines running atop the physical machines. Each ESX/i server is known as a **host** in vSphere. ESX/i hosts can be grouped together into a vSphere **cluster**. Grouping ESX/i hosts into clusters allow their computing resources to be aggregated together into a pool. Once a cluster of hosts is available, the available resources can be partitioned into resource pools. Each resource pool then has its own configuration settings on how resource allocation and reservation should be handled.

All disk arrays used to provide storage are known as **datastores** in vSphere. An ESX/i host can use a datastore after the datastore has been added to the host. There may be several hosts who are using the same datastore.

Having a resource pool enables the **provider virtual data center** (provider vDC) abstraction in vCloud Director. The service provider sets up one or more provider vDCs in vCloud Director to function as a data center that provides resources to all customers. When setting up the provider vDC, the service provider must also select some datastores to serve as storage spaces for all virtual machines that are to be run in this particular provider vDC.

Customers of the service provider in vCloud Director are known as **organizations**. Each organization generally represents a group of users with their own private control policies and running virtual machines. Members of the organizations cannot view the provider vDC, which is an abstraction that is only available to the service provider. Thus they also cannot allocate resources directly from a provider vDC to create a vApp. Instead, the service provider must explicitly allocate **organization virtual data centers** (org vDC) for each organization, and specify an appropriate amount of computing and storage resources along with the allocation model (resource commitment and reservation) according to the service level agreements (SLA) between the service provider and the organization. An organization's org vDC presents all the resources that are available to the members of the organizations. Each organization is allocated their own org vDCs, and is unable to view the org vDCs of other organizations. Org vDCs of different organizations may actually be backed by the same provider vDC. However, the organizations do not know of each other's resource

usage, resource reservation, and other sensitive information.

Virtual machines are thus deployed on org vDCs in vCloud Director. The end user does not know or care about the particular physical server that their virtual machines are run on. A virtual machine will be run on a particular physical server in the cluster, and may be moved dynamically to other physical servers from time to time depending on the different allocation policies set by the service provider and the current load on each physical server.

For more information on vSphere, please see VMware's documentation for vSphere [6]. And for more information on vCloud Director, please see VMware's documentations for vCloud Director [7] [8].

2.1.2 VMware vCloud Director Entities

VMware vCloud Director introduces numerous abstractions designed to enhance the cloud experience for both the service provider and the cloud consumer. These abstractions, some of which had been previously discussed, are known as **cloud entities**, and the combination of cloud entities and their configuration on vCloud Director constitutes a **cloud state**, which can be thought of as a snapshot of a cloud at some point in time.

In this section we discusses the entities in vCloud Director that are directly supported by vCloud Populator. Note that vCloud Populator does not currently support networking, and as such this thesis does not provide any information on networking in vCloud Director. Please refer to VMware's documentation on vCloud Director [7] [8] for more information on vCloud Director networking.

The vCloud Populator supports direct creation of all entities listed below, except for the provider vDC entity and the role entity.

Provider Virtual Datacenter (Provider vDC)

The service provider must set up all at least one provider vDC in his vCloud Director for it to be usable by his customers. He does so by selecting a vSphere resource pool

to be used to provide all computing and storage resources to back this provider vDC. The provider vDC is the virtual data center of the service provider. It is not directly accessible to the service provider's customers.

The vCloud Populator currently does not support the creation of a new provider vDC. All vCloud Director instances to be used with the vCloud Populator must already possess at least one provider vDC.

Organization

An organization is a customer of the service provider. If the service provider is a third-party cloud computing provider, each organization is likely to be an enterprise who is a billing customer of the service provider. On the other hand, if the service provider is the information technology (IT) department of a company, each organization may represent a different department in the company. The organization abstraction represents a grouping of users along with their own virtual data centers and their own policies.

The organization abstraction allows the service provider to host many different customer organizations in the same cloud with an isolation mechanism in place to ensure security and privacy. All actions performed by a user is only visible to his organization and should not affect other organizations.

Organization Virtual Datacenter

A provider vDC belongs to the service provider, and is not accessible to users of a customer organization. Therefore, the service provider must create org vDCs for each organization. The org vDCs are backed by the resources in a provider vDC. Several org vDCS, possibly belonging to different organizations, may be backed by the same provider vDC. Therefore, the org vDC can be seen as a partition of the provider vDC. When creating an org vDC, the service provider must decide on the allocation model of the org vDC. The allocation model determines how resource allocations work in this org vDC. The possible allocation models are **allocation pool**, **reservation pool**, and **pay as you go**. For more details on the different allocation models, please see

VMware's vCloud Director documentation [7] [8].

Once an org vDC has been allocated to an organization, members of the organization can now deploy their virtual machines to run on the org vDC's resources.

Virtual Application (vApp)

A collection of virtual machines that work together as an application to provide different kinds of services is known as a virtual application (vApp). Each vApp contains zero or more virtual machines. A vApp records information about the virtual machines' collaborations, such as any specific start up order when powering on the vApp, any sorts of networks that connect the virtual machines, etc.

One way to create a vApp on vCloud Director is to compose one from scratch. Using the web portal, a user can add blank virtual machines to a vApp. These blank virtual machines have no operating systems, let alone applications. Creating a vApp this way requires the user to manually set up the vApp and install the appropriate operating systems and applications on the individual virtual machines after the vApp is created.

Virtual Application Template (vApp Template)

Most of the time, working vApps are created by instantiating a vApp template. A vApp template, similar to a vApp, contains a collection of zero or more virtual machines. However, vApp templates are generally immutable and cannot be created from scratch. They also cannot be deployed. They serve as a "frozen copies" of a vApp at some point of time. Generally, vApp templates are considered preconfigured vApp images with the appropriate operating systems and applications already installed on the virtual machines, so that the configuration process can be skipped when creating a new vApp. VApp templates are stored in org vDCs, utilizing the available storage resources. A vApp template can be created either by capturing an existing vApp, or can be uploaded to vCloud Director directly from disk.

Media

Media represent virtual external media source, such as an ISO file or a floppy disk, and can be attached to running vApps, similarly to how users connect floppy drives and CD drives to a physical computer. Media are generally immutable. They are stored in org vDCs, utilizing the available storage resources. Media are generally uploaded to vCloud Director directly from disk.

Catalog

A catalog is a centralized place containing references to vApp templates and media from different org vDCs. vApp templates and media can optionally be added to a catalog to allow for easier access. Catalogs allow an organization admin to control access to vApp templates and media. VApp templates and media can be made available through sharing and publishing.

Role

The role of a user determines what actions the user can perform. A role is associated with a combination of rights. A right is a privilege to perform a certain action. An example right be "delete a vApp", and any users with a role that contains this right are allowed to delete vApps. Rights are all predefined in vCloud Director. vCloud Director comes with a list of default roles, each with their combination of rights. For example, the system administrator role (which should be given only to a user representing the service provider) combines all available rights, thus a user with the system adminstrator role can perform all possible available operations.

The list of predefined roles are the system administrator, the organization administrator, the catalog author, the vApp author, the vApp user, and console access only. New roles can be defined, but the vCloud Populator currently does not support the creation of new roles.

User

A user belongs to an organization, and has the ability to perform different actions in the organization based on his/her role. Each user can have only one role, which specifies the privileges that the user have. The user abstraction includes the user's login credentials, along with other information about him/her, such as his/her instant messenger, telephone number, etc.

2.1.3 vCloud Populator-specific Terms

As seen from the previous section, vCloud entity abstractions have a built-in structural hierarchy. For example, an org vDC is "based on" both an organization and a provider vDC. A user belongs to an organization. Media and vApp templates are stored in an org vDC but can both optionally be added to a catalog. Each of these entities has a dependency on higher-level entities: The entities simply cannot be created unless we know which higher-level entity they are based on. For example, we cannot try to create an org vDC unless we know which organization we are creating it for, and which provider vDC will be providing the necessary backing resources. In that case, we call both the organization and the provider vDC a **parent** of the org vDC. From a hierarchical point of view, the provider vDC and the organizations are "higher up" in the graph. The org vDC is thus a **child** of both the organization and the provider vDC.

There is a special type of parent, known as the **owner**, for every entity. The owner of an entity intuitively "owns" the entity, whereas non-owning parents simply are "referenced" by the entity. The perfect example for the distinction is the user entity. The user entity has two parents: the organization and the role. The user belongs to an organization, but it must know what role it has because it determines the actions that the user can perform. The organization entity is the owner of the user entity, because the user is a part of the organization. It would not have made sense for the role entity to be the owner of the user entity. A role should only know about what privileges it represents, but not about the users that reference it.

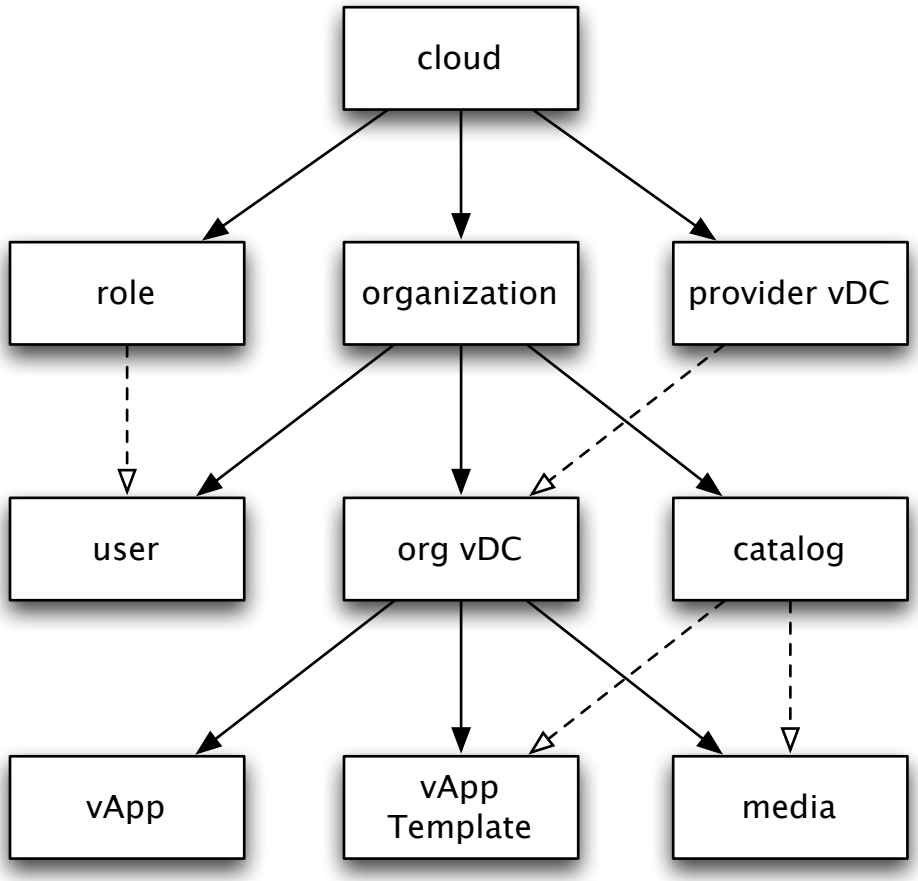

Figure 2-2: Displays the vCloud Director entities' relationships. A solid line from one entity to another means that the first entity is an owner of the second, while a dashed line from one entity to another means that the first entity is a non-owning parent of the second.

Every entity has at least one owner, and it may also have zero or more non-owning parents. Figure 2-2 shows the vCloud Director abstraction model as used by vCloud Populator.

An entity possesses many configurable **attributes** that customizes it. An example of an attribute is the name of an organization, or the allocation model of an org vDC. An entity's attributes, along with its parents, is known as the entity's **configuration**. Each attribute or parent in an entity's configuration is known as a **property**. Finally, an object's configuration together with its children is referred to as the entity's **state**.

To see the complete listing of information about an entity's properties and children, please see Appendix B. Note that the terms introduced in this section is only used specifically in vCloud Populator.

2.2 Domain-Specific Languages (DSL)

Domain-specific languages (DSL) are programming languages created specifically for a certain problem domain. When designed properly, they tend to be much more intuitive to read for domain experts than general-purpose programming languages, since any unneeded language constructs that might baffle non-programmers are stripped away. For our problem domain, cloud population, we choose a design with language constructs that focus on cloud state declaration, with an emphasis placed on the structural hierarchy of vCloud Director.

For further reading on designing domain-specific languages, please see Martin Fowler's "Domain-Specific Languages" [9].

2.3 Groovy

Our domain-specific language is designed based on the Groovy programming language. Groovy [10] is an object-oriented dynamic programming language that compiles to JVM bytecode. Therefore, Groovy code is generally compatible with most Java code. Using Groovy for our domain-specific language means that the result-

ing scripts can run and interact directly with existing Java programs, which is very helpful as most of our developers are coding in Java.

Groovy's basic syntax is very similar to Java's, but it has a plethora of new syntax and features that make it a suitable choice for designing domain specific languages. Please refer to Fergal Dearle's "Groovy for Domain-Specific Languages" [11] for further reading on how to create domain-specific languages with Groovy. We will now discuss some of the relevant features that are used in our language design. In Chapter 3 we will explain how our domain-specific language uses these relevant Groovy features to simplify its design.

2.3.1 Closure

A `Closure` is a block of code that can be executed upon invocation. A `Closure` in Groovy is declared within curly braces.

```
def closurePrint = {
    arg ->
    println arg
}

...
println "Hello World"
closurePrint("Executing closure")
```

Listing 2.1: Example of how to declare a `Closure`.

In Listing 2.1, a `Closure` named `closurePrint` is defined. Inside of the curly braces, a block of code can be declared. In our example, our `Closure` takes an argument named `arg` and prints it. Arguments to `Closure`s, as shown, are declared by placing them before an arrow (`->`). Note that closures are executed at time of invocation, not at time of definition. Therefore, `"Hello World"` will actually be printed before `"Executing closure"`, since `closurePrint` was invoked last.

`Closure`s can be passed around as variables, and therefore, they can be passed as arguments into methods, as shown in Listing 2.2.

```
void executeClosure(Closure c) {
    doSomething(); // random code to do something
    c();
}

def myClosure = { println "Hello World" }

executeClosure(myClosure);
```

Listing 2.2: This Listing demonstrates how to passing a `Closure` to a method as an argument.

If a method's last argument is a `Closure`, the method invocation is allowed to place an explicit `Closure` definition outside of the argument parentheses. This is only allowed if the `Closure` is defined at that point. We cannot pass in a `Closure` that was already defined earlier outside of the argument parentheses.

```
void executeClosure(int num, Closure c) {
    ... // do something with the closure and the number
}

// valid: Defining Closure
executeClosure(3) { println "Hello World" }

// valid: Closure can be defined inside argument parentheses
executeClosure(3, { println "Hello World Again" })

def myClosure = { println "More Hello World" }

// valid
executeClosure(3, myClosure)

// invalid: Cannot pass in a Closure that is already defined
executeClosure(3) myClosure
```

Listing 2.3: This Listing demonstrates valid and invalid ways of passing a `Closure` to a method as the last argument.

Listing 2.3 shows an example of a method that has a `Closure` as its last argument, along with different ways to invoke the method. The last method invocation syntax is

not valid, since it was not passing in an explicit `Closure` definition but still attempted to place the `Closure` outside of the argument parentheses.

For more information about Groovy `Closure`s, see Groovy's documentation on `Closure`s [12].

2.3.2 Named Arguments

Groovy methods allows the usage of **named arguments** in method invocations, as shown in Listing 2.4. In this example, `myMethod` is invoked with 5 arguments, 3 of which are named. Groovy will combine all named arguments into a `Map`, and pass this `Map` (which we will refer to as the **named arguments map**) as the first argument to `myMethod()`. All unnamed arguments will be passed to the method in the order in which they were listed. Therefore, `number1` will be passed the value 5 and `number2` will be passed the value 7.

```
void myMethod(Map myMap, int number1, int number2) {
    // myMap can be a map of named arguments
}

myMethod(a:3, b:4, 5, c:6, 7)
```

Listing 2.4: This Listing demonstrates how named arguments work in Groovy.

Note that the named arguments map will always be passed into a method as the first argument. Therefore any method that anticipates named arguments should expects the first argument to be a `Map`.

For more information about using named arguments in Groovy, see Groovy's documentation on named arguments [13].

2.3.3 Metaclass

All classes that are used in Groovy, be they Groovy or Java classes, have a **metaclass** associated with them. The metaclass of a class allows methods and fields to be added

to the class dynamically.

```
Integer.metaClass.myPlus = {
    other ->
    return delegate + other
}
Integer.metaClass.myMinus = {
    other ->
    return delegate - other
}
```

Listing 2.5: This Listing demonstrates how to add a method to a class through its metaclass.

Listing 2.5 dynamically adds a `myPlus()` method and a `myMinus()` method to the `Integer` class. The word `delegate` is a Groovy keyword, and in this context it refers to the object that the method is being added to.

2.3.4 Overwriting `methodMissing()`

Another way to dynamically add methods to an existing object is through overwriting the object's `methodMissing()` method. The `methodMissing()` method is invoked by Groovy whenever a method invocation has occurred, but Groovy cannot find the specified method. If the `methodMissing()` method is not defined, then Groovy will throw a `MethodMissingException`.

The `methodMissing()` method has two arguments: the first being the name of the method that cannot be found, and the second is an array of arguments to the missing method.

Listing 2.6 shows how to rewrite the example from the previous section by overwriting `methodMissing()`. In this example, the method `myPlus()` and `myMinus()` has both been added dynamically to the `Integer` class by overwriting `methodMissing()`. When `3.myPlus(4)` is invoked, Groovy will not be able to find the method `myPlus()` in the `Integer` 3 object, in which case it will invoke `methodMissing()` on `Integer` 3, with the arguments `"myPlus"` (a `String`) and an array whose only element is `4`.

```
class MyIntegerHandlerClass {
    static Integer myPlus(int firstNum, int secondNum) {
        return firstNum + secondNum
    }
    static Integer myMinus(int firstNum, int secondNum) {
        return firstNum - secondNum
    }
}

Integer.metaClass.methodMissing = {
    name, args ->
    return MyIntegerHandlerClass."$name"(delegate, args[0]);
}

3.myPlus(4)
```

Listing 2.6: This Listing demonstrates how to add a method to a class by overwriting its `methodMissing()` method.

`methodMissing()` will then call the static method `MyIntegerHandlerClass.myPlus()` with the arguments `3` (as represented by `delegate`) and `4` (as represented by `args[0]`). The `myPlus()` method of `MyIntegerHandlerClass` will add the two numbers then return the result. Note that error handling is left out of this example for simplicity.

Overwriting `methodMissing()` may seem less straightforward than simply adding a method directly to the class through the metaclass, however it is very useful when there are numerous methods to be added that are already defined elsewhere. For the vCloud Populator, we generally add methods through overwriting `methodMissing()`.

2.3.5 `Script`, `GroovyShell`, and `Binding`

Any script file written in valid Groovy can be converted into a Groovy `Script` object, which can then be executed directly inside a Groovy program by invoking its `run()` method. To convert a `java.io.File` into a `Script` requires the use of a `GroovyShell`. The `GroovyShell` has the ability to parse a `java.io.File` to determine whether it contains valid Groovy code, and if so converts it into an executable Groovy `Script` object.

The GroovyShell can also pass variables into a Script object through the use of a Binding. The Groovy Binding class contains a collection of variables and their values. Throughout the execution of the Script, any newly defined variables will be added to the Binding.

```
Binding binding = new Binding();

// Our own method to add variables to binding.
addVariablesToBinding(binding);

GroovyShell shell = new GroovyShell(binding);
Script dslScript = shell.parse(file);

// Run the script. Script has access to all variables.
dslScript.run();

/* Examine binding. Binding should now have all variables
 * introduced in the dslScript. */
examineBinding(binding);
```

Listing 2.7: This Listing demonstrates how the GroovyShell, Binding, and Script classes work together.

Listing 2.7 shows how to use all three of these classes together to run a Groovy script file while gaining access to the variables used by the script. We are free to add whatever desired variables to a Binding object. After passing the Binding to the GroovyShell constructor, all new Script objects returned from the GroovyShell's parse() method will use the Binding to store any variables in use by the Script. Thus, if a new variable is introduced in the Script, they will be automatically added to the Binding object. After the script has finished executing, we can now examine the Binding to see what new variables were used.

This feature in Groovy will prove very useful in our parser design, as will be discussed in Chapter 4.

2.4 Visitor Pattern

The Visitor Pattern [14] is a software design pattern that allows the definition of new operations on objects without changing the object classes. It allows for the separation of object models and algorithms.

The object models in the Visitor Pattern to which new operations should be added are known collectively as **Elements**. Each type of object model is represented by an implementation of the Element class, known as the **ConcreteElement**. The Element class has an `accept` method which takes a **Visitor** as an argument, and all ConcreteElement classes must provide implementations for the method. A Visitor represents a new operation, and is implemented by the **ConcreteVisitors**. Each type of new operations are represented by individual ConcreteVisitors. The Visitor class must declares a `visit()` method for each type of ConcreteElement.

Any program can now add new operations to any number of ConcreteElements by creating new ConcreteVisitors, then invoking the particular ConcreteElement's `accept()` method with an instance of the new ConcreteVisitor as an argument. The `accept()` method of all ConcreteElements should be implemented by calling the appropriate `visit()` method of the Visitor class.

The Visitor Pattern is used in the vCloud Populator system implementation, as will be discussed in Chapter 4.

2.5 Concurrency

To allow for parallelization of the vCloud Populator, we make use of some well-known concurrency constructions, such as the producer-consumer pattern using the Java's BlockingQueue and task execution using Java's Executor. This section will discuss these concurrency patterns and constructions. For further reading on these concurrency constructions in Java, see Brian Goetz et al's "Java Concurrency in Practice" [15].

2.5.1 Producer-Consumer Pattern using `BlockingQueue`

The Producer-Consumer Pattern allows for the decoupling of producers (classes that generate data) and consumers (classes that consume data), allowing for producers and consumers to proceed at their own pace without the need for complex synchronizations. The "data" in question depends on the use of the pattern.

BlockingQueue interface specifies a type of queue that offers several ways of handling `insert` and `retrieve` operations that cannot be executed immediately for one reason or another. We are mostly interested in the blocking approach, which provides a blocking `put()` method to insert new objects into the queue and a blocking `take()` method to retrieve and remove new objects from the queue. If there is no space in the queue, the `put()` method will block until space is available (presumably because another `Thread` had removed some objects from the queue). If the queue is empty, the `take()` method will block until an object becomes available (after another `Thread` inserts a new object into the queue).

2.5.2 Task Execution using `Executor`

`Executor` [16] is an interface declared in `java.io.concurrency` that is designed to manage and perform task execution. Task submission and execution are decoupled, with the actual task execution managed by the implementation of the `Executor` interface. The `java.io.concurrency` package provides a number of implementations for `Executor`. The `Executor` interface only declares one method, `execute(Runnable)`, which executes the supplied task, in `Runnable` [17] form, some time after this method is invoked.

The `Runnable` interface declares a single method, `run()`, which is expected to be implemented with code to be executed. In the case of submitting tasks to the `Executor`, the user-defined tasks are basically implementation of the `Runnable` interface that implements the `run()` method with the block of code to be executed for that particular task. A `Runnable` cannot return a result and cannot throw checked `Exception`s.

```
package java.lang;

public interface Runnable {
    void run();
}
```

Listing 2.8: This Listing shows the Interface for `Runnable`.

Many applications that make use of this task execution mechanism may want their task to compute a result or to throw checked `Exception`s. In that case, the applications will create a `Callable` [18] instead of a `Runnable`.

```
package java.util.concurrency;

public interface Callable<V> {
    V call() throws Exception;
}
```

Listing 2.9: This Listing shows the Interface for `Callable`.

However, we cannot submit a `Callable` to an `Executor`, since the `Executor`'s `execute()` method only takes in a `Runnable` as an argument. Java provides a way to submit `Callable` by introducing the `ExecutorService` [19] interface. The `ExecutorService` interface extends the `Executor` interface, declaring three `submit()` methods (with different arguments) that allows the submission of a task in either `Runnable` or `Callable` form. It also adds a number of convenience methods for invoking multiple tasks at once and some methods to deal with executor termination.

The manner in which tasks are executed by the `ExecutorService` is based on the implementation. Java includes several standard implementations. The one we will use in the vCloud Populator is the `ThreadPoolExecutor` [20]. The `ThreadPoolExecutor` works with a pool of `Thread`s. New submitted tasks are executed on any available `Threads`. If all current `Thread`s are busy, the `ThreadPoolExecutor` may create a new `Thread` for a submitted task, or it may queue the task to be executed later when a `Thread` has become available. This is all dependent on how the `ThreadPoolExecutor` is configured.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

vCloud Specification Language

This chapter discusses the **vCloud Specification Language** (vCSL). vCSL is a declarative language designed for rapid declaration of any cloud state. This chapter is broken down into several sections, each of which describes a main feature or functionality in the vCSL, along with its syntax. Note that the vCSL introduces many keywords for both cloud entities and their attributes. For a complete list of keywords and how they used be used, please refer to Appendix B.

vCSL is an extension to the Groovy language and as such, all vCSL scripts are valid Groovy scripts. Although not encouraged and not directly considered a part of the vCSL, users writing vCSL scripts can make use of built-in Groovy programmtic constructs such as loops and if-else statements in their scripts.

3.1 Entity Declaration

One of the most important piece of information captured in a cloud state is information about the cloud entities. Therefore, we must first and foremost introduce the mechanism for declarating an entity in a cloud state. An entity declaration begins with an **entity keyword**. There is one entity keyword for each cloud entity that is supported by the vCloud Populator. An example entity keyword is `org`, which is the keyword for the organization entity.

A entity declaration is signified by a set of parentheses that follow the entity

keyword, quite similarly to how methods in many mainstream languages are declared.

```
org()
```

Listing 3.1: Organization Declaration Example

Listing 3.1 shows an example of an entity declaration. It declares an organization without specifying any properties in its configuration. For example, even the name of the organization, which is required to be unique in vCloud Director, is not specified. This declaration is completely acceptable and valid. This is because vCSL provides default values for most attributes of an entity, including attributes with uniqueness requirements. This is known as the **basic default mechanism**. This can very helpful for vCSL users, since many users are only concerned about one or two properties in the entire configuration. This saves the users time from specifying and typing out unimportant properties.

Note that default values are not generated for all properties. The parents of an entity, for example, must be specified for every entity. Some attributes also must be specified, as listed in Appendix B.

3.2 Variables

Variables allow the association of a particular word to a value for later references. This value can be anything, from an entity to an attribute. The variable can be used not only within the script, but also in any other vCSL scripts that have included the current script or any Java/Groovy projects interacting with the vCSL parser. We will discuss this inclusion mechanism later in this chapter.

Usage of variables in vCSL is the same as in most mainstream programming languages, beginning with the variable name, then an equal sign ("="), and finally the entity declaration. The following example shows how to declare an organization, and then save the reference of the organization to a variable named `org_vmware`.

All variables have global scope. As such, even variables that have been declared

```
org_vmware = org()
```

Listing 3.2: Simple Variable Definition

in an external file are available for use if the external file has been imported. Also note that variable naming rules follow the same rules used by Java and Groovy.

3.3 Entity Configuration Specification

Simply being able to declare an entity is not enough to express an entire cloud state. Entities have many properties that must also be declared. This section discusses the numerous ways to capture the configuration of any entity.

3.3.1 Simple Attributes and Parents Specification

Default values for the attributes are extremely useful in situations where the user does not care about specific settings of an entity. However, users often do wish to specify some properties in the configuration. For example, they might wish to explicitly name their entities instead of generating the names through the basic default mechanism. The users also need to set the parents of the entities that they are declaring. We must provide a way for the user to specify some, but not necessarily all, properties of the entities' configurations. The vCSL allows for this via the use of **named arguments**, as introduced in the previous chapter. If an entity should have any specific properties, they must be passed into the entity declaration as named arguments, within the entity declaration parentheses. Each argument key in this case is a **property keyword**, and every entity has its own list of property keywords.

Listing 3.3 shows an entity declaration, where an organization with the name "VMware" and the full name "VMware Inc." is declared. Any attribute that has not been specified (for example, `description` in this case) does not have a specific value. The basic default mechanism will be activated during parsing, and default values will be generated for the unspecified attributes.

```
org(name:"VMware", fullname:"VMware Inc.")
```

Listing 3.3: Simple Entity Declaration

Parents, similarly to attributes, can be specified as named arguments. The argument key in this case is the entity keyword, and the value is a variable containing a reference to a declared entity. Listing 3.4 shows how to specify the parent organization of a catalog.

```
org_vmware = org(name: "VMware")
catalog = catalog(name: "operatingsystems", org: org_vmware)
```

Listing 3.4: Entity as Attribute

Properties do not have to be specified in a particular order. However, note that each property should only be specified once.

The values being passed in as arguments are generally the basic, well-known fundamental types in Groovy and Java, such as integers, strings, and boolean. Although vCSL is not strongly typed as a result of being Groovy-based, it is expected that the end user will pass in the correct type for each attribute.

3.3.2 Complex Attributes Specification

Some attributes are complex: they have their own attributes. For example, most of the organization's configuration are actually attributes of the organization's `settings` attribute. And the `settings` attribute has more complex attributes, such as the `generalSettings` and the `vAppLeaseSettings`. This can potentially cause some very complicated declarations as we traverse recursively down the attributes hierarchy. To represent a complex attribute, we employ the use of square brackets to group together all specified children attributes of a complex attribute.

Listing 3.5 demonstrates this. Here we can see that the organization declaration begins with the `org` keyword, followed by the open parenthesis just like any normal entity declaration. The first attribute that was declared is the `settings` attribute.

```
org (
    settings: [
        generalSettings: [canPublishCatalogs:true],
        vAppLeaseSettings: [deleteOnStorageLeaseExpiration:false]
    ],
    enabled: true
)
```

Listing 3.5: Complex Entity Attibutes

The value that is passed into the `settings` attribute is placed inside the square brackets: it includes two attributes, `generalSettings` and `vAppLeaseSettings`. Each of these attributes are complex in themselves, and require another pair of square brackets to denote that. As we can see, the `generalSettings` has a simple attribute named `canPublishCatalogs`, and the recursion ends there. `vAppLeaseSettings` similarly has only one simple attribute: `deleteOnStorageLeaseExpiration`, and the recursion also ends there. Finally, after the end of the `settings` attribute declaration, we have another attribute, `enabled`, which is a simple one. This example shows how attributes can be recursively declared in vCSL to allow for a clearer specification. Originally we had wanted to flatten the attributes hierarchy for simpler classification, however we decided to utilize this recursive attribute declaration syntax, since this approach is more similar to the vCloud Director abstractions and concepts, while providing good readability with its explicit structuring.

3.3.3 Attribute Reuse

All attributes are automatically upgraded to variables that are qualified by the entity that they belong to. For example, the `settings` attribute of an organization saved in the variable `org1` can be referred to as `org1.settings`. Listing 3.6 shows how to reuse qualified attributes from one organization declaration in another one. As we can see, `org2` is declared with using `org1`'s attributes, allowing for a level that reuse that can save many lines of code, especially if the complex attributes have many specified values.

```
org1 = org (
    settings: [
        generalSettings: [canPublishCatalogs:true],
        vAppLeaseSettings: [deleteOnStorageLeaseExpiration:false]
    ],
    enabled: true)

org2 = org (settings: org1.settings, enabled: org1.enabled)
```

Listing 3.6: Reuse of Attributes

3.4 Hierarchical Declaration

As discussed in Chapter 2, a vCloud Director state is structured hierarchically. The vCSL must be able to capture all hierarchical information about a cloud state. We had already introduced one way of capturing these hierarchical information: specifying parents of entities in named arguments. This specification style generally works well for small examples, but one can imagine that as files grow larger, it becomes increasingly difficult to trace through a vCSL script to determine the proper hierarchical information just from reading the file. Also, one parent entity may have many children entities, and having to list the same parent entity repeatedly in all children entities may be frustrating for the user.

To deal with this, we introduce a new syntax for declaration hierarchical information between an owner and its child. Note that this syntax does not work with non-owning parents and children entities. This syntax uses curly braces to signify the hierarchical ownership, as shown in Listing 3.7.

```
org_vmware = org(name: "VMware") {
    catalog_vmware = catalog(name: "VMwareLibrary")
}
```

Listing 3.7: Hierarchical Declaration

Any declarations that exist within the boundaries of these curly braces is considered a part of the owner entity. There is no longer a need to declare the owner

organization dependency in the child catalog.

Please keep in mind that any declared variables have global scope, and therefore any variables that are declared within the curly braces are still available for reference outside of the curly braces scope.

For entities that have non-owning parents, the non-owning parents must still be declared through the use of named arguments, as shown in Listing 3.8.

```
pvdc_main = pvdc(name: "main-pvdc")
org_vmware = org(name: "VMware") {
    orgvdc(name: "vCloud", pvdc: pvdc_main)
}
```

Listing 3.8: Listing Non-owning Parents in Named Arguments

3.5 Functional Declaration

Functional declarations help facilitate the scripting of a desired vCloud state.

3.5.1 The `common` Keyword

```
pvdc_main = pvdc(name:"main-pvdc")
org_vmware = org(name:"VMware") {
    orgvdc(name: "vCloud", pvdc: pvdc_main)
    orgvdc(name: "site-recovery-manager", pvdc: pvdc_main)
    orgvdc(name: "vcenter", pvdc: pvdc_name)
    orgvdc(name: "workstation", pvdc: pvdc_main)
    orgvdc(name: "fusion", pvdc: pvdc_name)
    orgvdc(name: "cloudfoundry", pvdc: pvdc_name)
    ...
}
```

Listing 3.9: Verbose Provider vDC Specification

The `common` keyword was introduced to simplify the declaration of similiar entities. For example, a user may find themselves writing the type of code demonstrated by

Listing 3.9 in their vCSL script. In this example the user is trying to create many org vDCs from the same provider vDC. It is a great hassle to have to rewrite `pvdc: pvdc_main` in every org vDC declaration. The `common` keyword is thus introduced to extract out common attributes so that they only have to be declared once. It is used similarly to an entity declaration: The declaration starts with the word `common`, followed by parentheses for a named arguments, and finally optional curly braces. The user should place attributes/parents that should be shared by several entities into the `common` keyword's named arguments. The entities that share these attributes/parents should be declared within the `common` keyword's curly braces.

```
pvdc_main = pvdc(name:"main-pvdc")
org_vmware = org(name:"VMware") {
    common(pvdc: pvdc_main) {
        orgvdc(name: "vCloud")
        orgvdc(name: "site-recovery-manager")
        orgvdc(name: "vcenter")
        orgvdc(name: "workstation")
        orgvdc(name: "fusion")
        orgvdc(name: "cloudfoundry")
        ...
    }
}
```

Listing 3.10: Use of `common` Keyword

Listing 3.10 demonstrates a simple usage of the `common` keyword. As we can see, the parent declaration `pvdc: pvdc_main` has been moved out of the org vDC's named arguments into the `common` keyword's named arguments. We choose to provide this keyword instead of allowing nested parent declaration (e.g. org vDC declaration nested inside a provider vDC declaration which is in turn nested inside an organization declaration) because 1) it explicitly informs the reader that a provider vDC is not a part of the organization, and 2) this usage will work for not only parent declaration, but also for any general attribute declaration.

The `common` keyword only applies to any entity declarations made in the top level of the `common` keyword's nesting. Take a look at Listing 3.11. The `pvdc: pvdc_main`

declaration applies only to the org vDC declaration, but not to the media declaration, since the media declaration is not made in the top level of the common nesting.

```
pvdc_main = pvdc(name:"main-pvdc")
org_vmware = org(name:"VMware") {
    common(pvdc: pvdc_main) {
        orgvdc(name: "vCloud") {
            media(file: "some-file.iso")
        }
    }
}
```

Listing 3.11: One Level of `common`

```
pvdc_main = pvdc(name:"main-pvdc")
org_vmware = org(name:"VMware") {
    common(pvdc: pvdc_main]) {
        common(description: "I am in group A!") {
            vcloud = orgvdc(name: "vcloud")
            srm = (name: "srm")
            vcenter = orgvdc(name: "vcenter")
        }
        common(description: "I am in group B!") {
            orgvdc(name: "workstation")
            orgvdc(name: "fusion")
            orgvdc(name: "cloudfoundry")
        }
        orgvdc(name: "zimbra")
    }
}
```

Listing 3.12: Nested Common

For convenience, the `common` keyword can be nested, as shown in Listing 3.12. In this vCSL script, all of the declared org vDCs will have a dependency on the `pvdc_main` because of the top level `common` keyword. The org vDCs `vcloud`, `srm`, and `vcenter` will all have the description "I am in group A!", while the org vDCs `workstation`, `fusion`, and `cloudfoundry` will all have the description "I am in group B!". The org vDC `zimbra`, however, will not have a description. As we can see, the top

`common` keyword's named arguments is carried down to its nested children, and will be added to the named arguments of all entity declarations. However, if another `common` keyword is encountered, the current named arguments will be added to the named arguments of this new `common` keyword, and will be carried on to the new `common` keyword's children. This nesting and addition of maps will carry on recursively until there are no more `common` declarations.

Note that although most of the syntax for the `common` declaration is quite similar to that of an entity declaration, the result of the common declaration cannot be saved into a variable: a compilation exception will be thrown if the user attempts that.

3.5.2 The `data` Keyword

The `data` keyword allows the user to provide a CSV (comma-separated values) file of default values for specific entities. The built-in basic default values mechanism is sufficient for many use cases, however some users may wish to populate their cloud with more realistic data. In that case, the user can specify the file path of a CSV file containing a collection of default data.

A default data file can contain default data for different kinds of cloud entity. One data file can contain many default data groups. Each default data group contains default data for one kind of cloud entity. Default data groups should be separated by an empty line.

Each default data group must begin with an entity keyword on a line by itself, declaring the type of cloud entity that this default data group is for. The second line in the default data group contains attribute names. The rest of the lines in the group contains default data values for each of the listed attributes. The user must take care not to have any empty lines in a default data group.

Look at the example in Listing 3.13 for a simple example of a CSV file used by this default data mechanism. If the user wish to set the default data for an attribute in a complex attribute, the attribute must be qualified, as shown in Listing 3.14.

The `data` keyword should only be used as a top-level declaration.

```
user
name,   fullname,   emailAddress,       telephone
asmith, Anne Smith, asmith@vmware.com,  (912)-784-2281
jdoe,   John Doe,   jdoe@vmware.com,    (201)-142-8820
...
```

Listing 3.13: Example CSV file

```
orgvdc
name, allocationModel, cpuCapacity:allocated, cpuCapacity:reserved
data....
...
```

Listing 3.14: Example CSV File With Complex Attributes

3.5.3 The `include` Keyword

The `include` keyword imports external vCSL scripts for use in the current script. Any entities that have been declared or any variables that have been defined in the external script is imported. The `include` keyword should only be used as a top-level declaration. Listing 3.15 shows how the `include` keyword can be used in a vCSL script to import an external vCSL script.

```
======================== Sample mylib.vcsl ========================

pvdc_root = pvdc(name: "Root")

============ Sample vCLSL script importing mylib.vcsl =============

include "mylib.vcsl"

org() {
    orgvdc(pvdc: pvdc_root);
}
```

Listing 3.15: Sample vCSL Script Using `include` Keyword

Note that the vCSL comes with a standard library file named `stdlib.vcsl`, which declares all the predefined roles in vCloud Director. See Appendix D for this file.

3.5.4 Numerical Declaration

The last functional declaration to be introduced is the numerical declaration. The numerical declaration is a declarative version of the for loop: it allows the users to declare multiple instances of the same entity in one line.

```
4.org()
```

Listing 3.16: Numerical Declaration

This declares four organizations in one line. This is very useful for cases where specific attributes are not important and the user just wishes to quickly script up a large cloud. Note that since this declaration has not supply any attributes, default values will be used. If the user has not provided a data file, these four organizations will most likely be provided with identical values (besides for attributes which has a uniqueness requirement, such as the name). If the user wishes to use the numerical declaration syntax to create non-identical entities, use the `data` keyword.

Chapter 4

System Implementation Overview

This chapter seeks to dive into the implementation details of the vCloud Populator blackbox. We will first introduce the many different components at work within this system. The vCloud Populator system is composed of two major components: the parser and the execution engine. Figure 4-1 shows the interactions between these two blackbox components. The first input to the vCloud Populator system is a vCSL script, which describes a desired output cloud state. The second input is the reference to the currently empty cloud. The vCSL script is analyzed by the parser, which constructs a cloud entity model graph that represents the cloud state. This output cloud entity model graph is then passed to the execution engine, along with the reference to the empty cloud. The execution engine now traverses the cloud entity model graph and makes the appropriate calls to vCloud Director to populate the cloud according to the cloud entity model graph.

4.1 Cloud Entity Model Graph

We would like to begin the system implementation overview by describing the **cloud entity model graph**. The cloud entity model graph is an algorithm-agnostic representation of a VMware vCloud Director state. As discussed previously, vCloud Director's abstraction model includes a rich hierarchy of entities. For cloud population, we know that certain entities must be created before others, establishing a

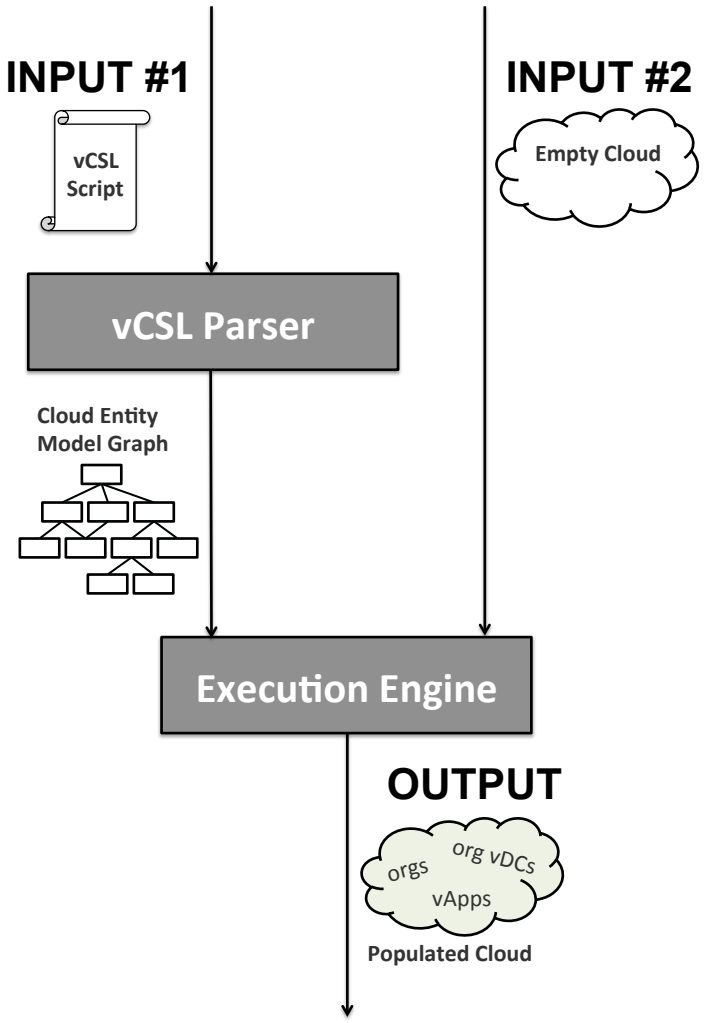

Figure 4-1: This diagram shows the interaction between the parser and the execution engine by showing the different inputs and outputs of the components and the overall system.

parent-child relationship in the process between entities that must be enforced. We simply cannot create an org vDC before an organization, for example. To accommodate that, we have decided to represent the abstraction model with the cloud entity model graph, a directed graph where each node represents one cloud entity and each directed edge represents a parent-child relationship. Although this representation is tailored for the cloud population case, we anticipate that this intuitive representation for vCloud Director's abstraction model will have many other uses. Therefore, we have tried to make the cloud entity model graph as generic as possible, in hope that it can be reused for other purposes. However the main focus of this section will be on the cloud population use case.

The cloud entity model graph, representing a vCloud Director state, is actually an intermediate product between the parser and the execution engine. The parser will generate the cloud entity model graph from a vCSL script, and pass the cloud entity model graph as input to the execution engine. The execution engine will then populate the cloud based on the cloud entity model graph. Currently, the cloud entity model graph is actually a collection of Java `CloudEntityModel` objects in memory, where each `CloudEntityModel` object acts as a node in the cloud entity model graph and children/parent-getters in these objects act as the directed edges. `CloudEntityModel` is a Java interface, and each type of entity in vCloud Director is represented by an implementation of `CloudEntityModel`. For example, an org vDC is represented by the `OrgVdcModel` implementation class, while a catalog is represented by the `CatalogModel` implementation class.

All `CloudEntityModel` classes have the following methods:

```
List<CloudEntityModel> getChildren();
List<CloudEntityModel> getParents();
void accept(CloudEntityVisitor visitor);
```

Listing 4.1: `CloudEntityModel` Methods

The first two methods are somewhat self-explanatory: They are simply getters for the parents and children of an entity. The third method requires a bit more discussion.

The third method is an `accept()` method in the Visitor Pattern. As explained previously, the Visitor Pattern allows separation of objects and algorithms. It is our wish that our `CloudEntityModel` objects can be generic enough to be used not only for cloud population but also for other purposes. The use of the Visitor Pattern allows the `CloudEntityModel` objects to be defined without knowing explicitly what type of actions will be performed on them. The basic idea behind the use of the Visitor Pattern in this case is to have all the `CloudEntityModel` objects implement an `accept()` method, which takes a *visitor* object as an input. Our visitor is this case is a `CloudEntityVisitor` object. The `CloudEntityVisitor` class has a `visit()` method for each type of `CloudEntityModel` objects (e.g. `OrgModel`, `UserModel`) and basically looks like Listing 4.2.

```
void visit(OrgModel orgModel);
void visit(OrgVdcModel orgVdcModel);
void visit(ProviderVdcModel providerVdcModel);
void visit(UserModel userModel);
...
```

Listing 4.2: `CloudEntityVisitor` Methods

It is up to each type of `CloudEntityModel` to implement the `accept` method properly by calling the correct version of the provided `CloudEntityVisitor`'s `visit` method. We will talk more about how the vCloud Populator uses the Visitor Pattern in its implementation in Section 4.3.2.

The `CloudEntityModel` interfaces and implementations are written completely in Java.

4.2 Parser

In this section we talk about the parser, written in a combination of Java and Groovy. The parser parses through a vCSL script and converts it into a cloud entity model graph. Here the word "parse" is used in a loose sense: The parser does not actually read in the script character-by-character, find the different tokens in the script, then

construct an abstract syntax tree from the tokens. As previously mentioned, the vCloud Specification Language itself is an extension to Groovy. Any scripts written in the vCloud Specification Language is technically a valid Groovy script. Thus, instead of redoing the work that the Groovy parser and compiler already do, our parser takes a different approach (one that many Groovy embedded domain specific languages take). It simply allows Groovy to run the vCSL script normally, until Groovy encounters a vCSL keyword that it does not know how to handle, in which case it invokes our parser to deal with the keyword appropriately. In some sense, our parser behaves very similarly to a keyword handler.

4.2.1 Parser Initialization

We start off the parsing process by passing in a vCSL script to a class named `ScriptParser`, which is the main entry point into the vCSL parser. `ScriptParser`'s main job is to initialize the parser. It does so by first converting the input vCSL script into a Groovy `Script` through the use of a `GroovyShell`.

As discussed previously, this `Script` object can be run by invoking its `run()` method, and Groovy will run it just like any normal Groovy script. The main challenge here, however, is that our vCSL script make uses of vCSL keywords, which Groovy does not inherently handles. By now the reader may have realized that all keyword declarations in the vCSL are basically method calls, and all properties are simply arguments to these methods. Because these method calls are not defined anywhere in the `Script`, nor are they built-in Groovy constructs, Groovy will quickly throw a `MethodMissingException` if we just run the `Script` as it is. Thus, an important task in the parser initialization is to inform Groovy to allow the parser to handle these unknown constructs. One of the easiest way to do so is to overwrite the `methodMissing()` method of our `Script` object to invoke a handler method defined by us. The `methodMissing()` call will pass the name of the method and its arguments to the handler method, and then it is up to the handler method to decide, on a case-by-case basis, what to do with each method call. Note that by overwritting `methodMissing()` of the `Script` object, Groovy will pass to our handler method *all*

method invocations that it cannot handle, which may include method invocations that are not vCSL methods, such as misspelled keywords. The handler method must be resilent enough to distinguish between correct vCSL code and programming errors.

Because there is a decent number of keywords, we decided to create subhandler methods for each keyword in our design, and have the handler method redirect execution to the correct subhandler based on the name of the input method, or simply throw a compilation exception if it does not recognize the input method name. Therefore, we have an entire handler class dedicated to the handler method and the subhandler methods. The class is known as the `ModelGraphBuilder`. The handler method is a method in the `ModelGraphBuilder` known as the `keywordHandler()`. Each subhandler method is named X`SubHandler`, where X is the keyword. As an example, the organization subhandler method is named `orgSubHandler()`.

Listing 4.3 shows how the `ScriptParser` class overwrites the `Script` object's `methodMissing()` method to redirect invocations of any unknown methods to the `ModelGraphBuilder`'s `keywordHandler()` method.

```
Script dslScript = ...;
ModelGraphBuilder builder = ...;

dslScript.metaClass.methodMissing = {
    name, args ->
    builder.keywordHandler(name, args)
}
```

Listing 4.3: Overwriting `Script`'s `methodMissing()` in `ScriptParser`

This handles all vCSL keywords. However, as the reader may remember, the numerical declaration does not have its own vCSL keyword. Instead, the syntax requires using an vCSL entity keyword as though it is a method of a number (`Integer`). Indeed, that is exactly how it is implemented in the vCSL parser. We again uses the `methodMissing()` overwriting technique to help us achieve this. Listing 4.4 shows us how to overwrite `methodMissing()` of the `Integer` class so that a construct like `3.org()` can be handled. The shown code will overwrite `methodMissing()` for all

Integers instead of for a specific object. Any time a construct like 3.org() is encountered, ModelGraphBuilder's numberSubHandler() method will be invoked, with the name of the entity (org), the args (none in this case), and the delegate (the actual object that the method was invoked on, which is the Integer 3) passed as arguments.

```
ModelGraphBuilder builder = ...;

Integer.metaClass.methodMissing = {
    name, args ->
    builder.numberSubHandler(name, args, delegate)
}
```

Listing 4.4: Overwriting Integer's methodMissing() in ScriptParser

After overwriting these two methodMissing() methods, the Script is finally ready to be run. In the next section, we will discuss what keywordHandler() and the subhandler methods actually do when they are invoked.

4.2.2 Keyword Declaration Processing

The keywordHandler() method is a straight forward method. It first checks whether the input method name is actually a keyword. If so, it invokes the appropriate subhandler method. If not, it throws a compilation exception. Listing 4.5 shows the implementation of keywordHandler() in pseudocode. For the remainder of this section, many examples will be shown in pseudocode to simplify the discussion and to avoid overwhelming the readers with unnecessary implementation details.

```
if method name is a keyword:
    return invokeMethod(name + "SubHandler", args)
else
    throw compilation exception
```

Listing 4.5: Pseudocode of ModelGraphBuilder's keywordHandler()

Basic Keyword Processing for Entities

What exactly is in the arguments that are passed into `methodMissing()`? In the simplest case, the named arguments into our entity declarations will be passed in as a `Map` as the sole argument. This `Map` thus contains all property declarations.

```
myorg = org(name: "VMware")

mycatalog = catalog(org: myorg)
```

Listing 4.6: Basic Entity Keyword Processing

In Listing 4.6, there are two method invocations. We will first examine the `org` invocation. For this invocation, the `orgSubHandler` will be invoked with the `Map [name: "VMware"]` as an argument. The orgSubHandler must record this declaration. It does so by creating an `OrgModel`. Remember from our cloud entity model graph discussion that there is one kind of `CloudEntityModel` implementation class for each kind of entity that is supported. Each `CloudEntityModel` class stores information about the configuration of that specific type of entities. It is thus the subhandler methods' jobs to create the appropriate `CloudEntityModel` object, and then fill in the `CloudEntityModel` object with the input properties. It must then return the created `CloudEntityModel`. In the case of this invocation, the returned value is saved into the variable named `myorg`.

Note that parents are considered a part of the configuration of an entity, and thus `CloudEntityModel`s must also contain references to all of their parents. The `catalog` invocation in Listing 4.6 demonstrates an example of this. For this invocation, the `catalogSubHandler` will be invoked with the `Map [org: myorg]`. The variable `myorg`, which is the created `CloudEntityModel` from the last invocation, is basically a reference to the parent entity. Therefore, the `catalogSubHandler` can simply save that into the created `CatalogModel`, and also inform the `OrgModel` that the created `CatalogModel` is now being added as a child. Then the `catalogSubHandler` can return the created `CatalogModel` so that it can be saved into the `mycatalog` variable.

The situation is a bit more complicated in the case of hierarchical declaration, as shown in Listing 4.7.

```
myorg = org(name: "VMware") {
    mycatalog = catalog()
}
```

Listing 4.7: Quick Hierarchical Example

By now, the reader may have realized that the curly braces used to signify structural hierarchy are actually the Groovy `Closure` construct. In fact, any method invocation followed by the `Closure` will be automatically passed the `Closure` as the second argument. A `Closure` is basically an executable block of code. When it is passed to the `orgHandler`, the `Closure` has not yet been invoked. And the `orgHandler` should not invoke the `Closure` until it has finished creating the `OrgModel` and filling in the configuration information.

At that point, the `orgSubHandler` is ready to invoke the `Closure`. The `Closure` will execute just like the `Script` object itself, and will eventually redirect execution to the `catalogSubHandler`. However, we have a problem: the parent `org` was not provided, so the `catalogSubHandler` will not be able to add the catalog to a parent organization.

To solve this issue, a `Map` known as the `environmentMap` was introduced. The `environmentMap` is a member variable of the `ModelGraphBuilder` class, so there should be one `environmentMap` per instance of the parser. Whenever a `Closure` is about to be invoked, the subhandler method adds the current `CloudEntityModel` to the `environmentMap`. After the `Closure` runs and returns control, the subhandler method removes the `CloudEntityModel` from the `environmentMap`. Therefore, at any point in time, the `environmentMap` should only contain entities that were declared directly above the current `Closure`. This allows a subhandler to retrieve parent entities by searching for it in the `environmentMap`.

Therefore, in the example, `orgSubHandler()` adds the created `OrgModel` into the `environmentMap` before invoking the `Closure`. The `catalogSubHandler` can thus

access this `OrgModel` and add the `CatalogModel` as a child. The `orgSubHandler` must remove the `OrgModel` from the `environmentMap` after the `Closure` has completed execution and returned control.

This concludes the basic explanation of the actions of a entity subhandler. Entity subhandlers will be revisited after discussions of the functional declarations to explore how entity declarations are integrated with functional declarations.

Subhandler for `common` Keyword

The first functional subhandler to be discussed is the subhandler for the `common` keyword. The `common` keyword, as the reader may recall, allows sharing of the `common` method call's property mappings between entities declared exactly one level under the `common` declaration. The `common` keyword is mainly implemented with another member variable in the `ModelGraphBuilder`: a `Map` known as `commonNamedArguments`.

```
commonSubHandler(namedArguments, closure) {
    add namedArguments to commonNamedArguments
    invoke closure
    remove namedArguments from commonNamedArguments
}
```

Listing 4.8: Steps Taken for `common` Keyword Subhandler

Listing 4.8 shows pseudocode for the steps that the `common` keyword subhandler takes. The subhandler must first add the input named arguments map to `commonNamedArguments`. This allows any one-level-deep declarations to use these arguments as their own. The subhandler then can invoke the `Closure`. After all the code in the `Closure` has finished executing, the named arguments that were provided by the input named arguments map should be removed from `commonNamedArguments`. Note that entries in `commonNamedArguments` should be intact after an execution of the `commonSubHandler`.

If we have multiple level of nesting of the `common` keyword, the named arguments will keep getting added to the `commonNamedArguments` until the first level of `Closure`

that is not a `common` declaration, so all of the named arguments that were added in the different levels will just be compounded and applied to that first non-`common` declaration.

Subhandler for `include` Keyword

The `include` keyword basically "imports" an external vCSL script by executing it and ensuring that any variables introduced in the external script is also present in the current script when control returns. It does so through allowing the two scripts to share the same `Binding` object.

```
includeSubHandler(filepath) {
    create GroovyShell with current Script's Binding
    use GroovyShell to create new Script from filepath
    set up new Script's delegation to ModelGraphBuilder
    run new Script
}
```

Listing 4.9: Steps Taken for `include` Keyword Subhandler

The `includeSubHandler` first creates a `GroovyShell` with the current `Script`'s `Binding`, then use that `GroovyShell` to create a `Script` object from the filepath of the external vCSL script. Now any variables that had been declared in the current `Script` so far are also available to the external `Script`, and any variables that are declared in the external `Script` will also be available to the current `Script` after the external `Script` is executed. The `includeSubHandler` must, of course, also set up the new external `Script`'s `methodMissing()` method to point to the `ModelGraphBuilder`'s `keywordHandler` method for the new `Script` to run correctly. Now the new external `Script` can finally be run.

Subhandler for `data` Keyword and the `DataDistributer`

The `data` keyword expects as an input the path of a CSV (comma-separated values) file. It then parses the file to obtain a collection of attribute maps, each of which is

then added to the DataDistributer. CSV file parsing is a trivial task and thus will not be discussed in details.

```
void dataSubHandler(dataFile) {
    parse through dataFile to obtain a collection of Maps
    add these Maps to the DataDistributer
}
```

Listing 4.10: Steps Taken for data Keyword Subhandler

The DataDistributer is a centralized place for all default data entries, and its main function is as its name suggests: it distributes the next available line of data to a requester. If there is no available data because either the data declaration was never used or that all available default data has been depleted, then the DataDistributer returns an empty attributes map, and the basic default mechanism kicks in. The DataDistributer has two methods: an addEntry() method to add a default data entry for an entity type, and a getNextEntry() method to retrieve the next available set of default data for an entity type.

Subhandler for Numerical Declaration

As discussed previously, the ScriptParser has also overwritten the methodMissing() method of the Integer class to delegate to the numberSubHandler() method of ModelGraphBuilder.

```
void numberSubHandler(entityName, methodArgs, numEntities) {
    invoke subhandler for the entity numEntity times
    save created CloudEntityModels into a List
    return the List
}
```

Listing 4.11: Steps Taken for Numerical Declaration Subhandler

The numberSubHandler will invoke the entity subhandler for the entity the given number of times. For example, if the method invocation was 4.org(), then the

numberSubHandler is invoked with org as the entityName and 4 as the numEntities. In that case, the numberSubHandler effectively invokes orgSubHandler() 4 times. Each time orgSubHandler() is invoked, it creates a new OrgModel, possibly with numerous children CloudEntityModels. All created OrgModels are then saved into a list, which is returned at the end of the numberSubHandler.

Integrating Functional Declaration with Entity Declaration

Now that we have discussed all the functional keyword subhandlers, we can finally revisit the entity keyword subhandlers. Listing 4.12 shows the complete pseudocode for any entity subhandler after integrating with the functional declarations.

```
CloudEntityModel entitySubHandler(namedArguments, closure) {
    if (attribute mappings empty)
        attempt to get data from DataDistributer
    create CloudEntityModel based on property mappings
    save and clear the current commonNamedArguments
    put created CloudEntityModel in environmentMap
    invoke closure
    remove created CloudEntityModel from environmentMap
    add back the commonNamedArguments' old entries.
    return created CloudEntityModel
}
```

Listing 4.12: Steps Taken for An Entity Keyword Subhandler

The first thing an entity keyword subhandler must do is to check whether or not the attribute mappings is empty. If it is, then it should attempt to retrieve data from the DataDistributer. This may succeed or fail based on whether or not any default data has been provided and whether or not the provided data source has been depleted. Regardless of the result, any attributes that ultimately do not have values will be filled in with generic default values by the basic default mechanism at the next stage: CloudEntityModel creation. After the creation step, it is time to clear the commonNamedArguments. This step is important to ensure that the common keyword functions properly, since the effects of common should apply only to entities

that are exactly one-level deep from where the `common` declaration is. But we have to be careful to save the entries we removed: any other declarations in the same `Closure` expect `commonNamedArguments` to be intact.

After clearing out the `commonNamedArguments`, the created `CloudEntityModel` can be placed in the `environmentMap`. Doing so allows children of the current entity to have access to it. At this point we can invoke the `Closure`.

After the `Closure` has completed execution, it is now time to undo what the entity keyword subhandler did. We must remove the created `CloudEntityModel` from the `environmentMap`, then add back the entries in the `commonNamedArguments` that was removed.

We can now return the created `CloudEntityModel`.

Sample Walkthrough

Here is a walkthrough of the steps for a small example.

```
pvdc_main = pvdc(name:"main")
org() {
    common(pvdc:pvdc_main) {
        orgvdc()
        orgvdc()
    }
}
```

Listing 4.13: Sample Walkthrough

1. `ScriptParser` initializes the `ModelGraphBuilder` class.
2. `ScriptParser` invokes `GroovyShell` to parse through the vCSL script `File` and return a `Script` object.
3. `ScriptParser` delegates all keyword handling to the `ModelGraphBuilder`. The `Script` object's `methodMissing()` method is overwritten to instead redirect to the `ModelGraphBuilder`'s `keywordHandler()` method, which determines the correct subhandler to dispatch the method call to, or throws an exception if the keyword is unknown. It also overwrites the `Integer` class's `methodMissing()` method to

redirect to `ModelGraphBuilder`'s `numberSubHandler()` method, which executes any numerical declaration correctly.

4. `ScriptParser` invokes the `Script`, which then runs as Groovy code.
5. `Script` encounters method call `pvdc(name:"main")`. Groovy cannot find such a method, and thus invokes the `Script`'s `methodMissing()` method with the two arguments: the `String "pvdc"` and the named arguments `Map [name:"main"]`.
6. `Script`'s `methodMissing()` redirects to `ModelGraphBuilder`'s `keywordHandler()`, which must does some basic setup for all keywords. The most important setup here is checking to see whether the `commonNamedArguments Map` is null. If not, all entries from the `Map` should be added to the attribute mapping to be used by the keyword subhandler also.
7. `keywordHandler()` can now invoke the provider vDC's keyword subhandler, `pvdcSubHandler()`, which creates the `ProviderVdcModel` with the given arguments. Because the `pvdc()` call has no input `Closure`, the subhandler has no further work to do, and can return the created `ProviderVdcModel`.
8. The created `ProviderVdcModel` is returned all the way up the levels of method calls, and is finally returned as the return value of `pvdc(name:"main")`. It is then saved in the variable `pvdc_main`.
9. The next line is now processed. Again, Groovy cannot find `org()`, and thus this method call is delegated to `keywordHandler()` in the `ModelGraphBuilder`, which in turns invoke `orgSubHandler()`.
10. This time, after creating the `OrgModel`, the keyword subhandler must also handle the `Closure`. It must now save the created `OrgModel` in the `environmentMap`, so that all entity declarations in the `Closure` have access to it. It must also clear the `commonNamedArguments` if it is not empty, since all common values should only apply to keyword declarations that are exactly one level lower than when the `common` keyword was last used. It can now invoke the `Closure`.
11. The `Closure` contains a `common(pvdc:pvdc_main)` declaration, which is ultimately processed by the `common` keyword subhandler, `commonSubHandler()`. The `common` keyword subhandler adds the input named arguments `Map`, in this case

[pvdc:pvdc_main], to the commonNamedArguments Map, then invoke the next level of the Closure (this time taking care not to clear commonNamedArguments).

12. In the Closure there is an orgvdc declaration. This declaration has no input arguments, but our handlers will add the commonNamedArguments as the properties mappings. Therefore, the org vDC keyword subhandler, orgvdcSubHandler(), now has access to both its parent organization (through the environmentMap) and its parent org vDC (through the properties mappings), and the OrgVdcModel can be created properly. The second orgvdc() call can be processed similarly.
13. By this point, the Closure has completed execution. Control now returns to the commonSubHandler(), which clears the commonNamedArguments Map before returning. The chain of keyword subhandler control return continues until we reach the top level again.

4.2.3 Error Handling

As discussed, not all constructs that are passed to the ScriptParser are actually correct. If a non-vCSL method is passed to the keywordHandler(), it will simply throw a compilation exception. The compilation exception is this case is the VCSLParserException. Note that we decide to throw this specific exception instead of the generic Groovy MethodMissingException to emphasize that the vCSL parser has detected an error. There can be exceptions that are thrown which are not VCSLParserExceptions. In that case Groovy has detected an error that the vCSL parser has not, and it may actually be a signal of an error in the vCSL parser itself. Having a separate compilation exception allows internal errors to be detected more easily. Therefore, we try to be consistent in our error detection and throw VCSLParserExceptions whenever we detect compilation errors.

A list of possible compilation errors that we anticipate is as follows:

1. Unknown keywords. ex. misspelling "catalog" as "catalg"
2. Unknown property. ex. misspelling "name" as "nae"
3. Unknown variables. ex. misspelling variable name after definition

4. Basic syntax error. ex. missing curly braces

When an error is encountered, the parser generally attempts to return the line number in the vCSL script where the error has occurred. This can be retrieved through the stack trace, whenever an exception is about to be thrown.

Unknown keywords

All unknown keywords are detected by `ModelGraphBuilder`'s `keywordHandler()` method. `keywordHandler()` simply checks the input keyword against a list of known vCSL keywords. If the input keyword is not a known keyword, then the parser throws a `VCSLParserException`.

Unknown properties

Because every type of entity has different kinds of attributes, we cannot simply take the same approach as the unknown keyword detection. Instead, we wait until the `CloudEntityModel` creation step. While we are filling in the `CloudEntityModel`, we can detect whether or not an attribute is not known to that particular type of entity. We can throw a `VCSLParserException` at that time.

Unknown variables

Just as Groovy invokes an object's `methodMissing()` method to handle any unknown method invocations on it, Groovy similarly invokes an object's `propertyMissing()` method when it encounters an attempt to access an unknown variable/property of the object. Any variable declared in the `Script` is considered a variable/property of the `Script`, and overwriting the `Script`'s `propertyMissing()` method allows our parser to intercept any unknown variable access. Since all vCSL keywords are technically Groovy methods, not variables, we know that if the `propertyMissing()` method is actually invoked, there must be an error, and a `VCSLParserException` can be thrown.

Basic syntax errors

Not all mistakes are caught by the vCSL parser. If a vCSL script has a very basic syntax error in it and is thus not even valid Groovy code (for example, unmatching number of parentheses), it will fail to compile properly. In that case, the end compilation error would be Groovy's, with the proper error message included. To provide a consistent interface, the parser will catch Groovy's compilation exception and rethrow the parser's own `VCSLParserException`.

4.3 Execution Engine

This section discusses the design and implementation of the execution engine. The execution engine, written in pure Java, is more straight forward in both design and implementation than the vCSL parser. Ultimately, the execution engine must process the cloud entity model graph, and for each `CloudEntityModel` invoke some vCloud API calls to a vCloud Director instance to create that particular entity on the server. The greatest challenges in designing the execution engine is to ensure that the vCloud API calls are made in the correct order so that all parent entities are created before their children are, and to design for concurrency.

4.3.1 Design

Before diving into the details of the implementation, we would first like to discuss the overall design of the system. The input into the execution engine is the cloud entity model graph. This ultimately means that the execution engine has a collection of objects to create, with certain constraints on the creation order placed by the fact that children cannot be created until all their parents are created. Each object creation can be considered a task, and is represented by a `CreationTask` class. Most `CreationTask`s are independent of each other and thus can be executed concurrently. We can utilize Java's `ExecutorService` to execute the `CreationTask`s for us in parallel. Any time an object is ready to be created, we can simply submit a request

to the ExecutorService, and the ExecutorService will arrange for the object creation. Because we do not have a good control over the order of the tasks' execution once they are submitted, we must be careful not to submit CreationTasks unless we are sure that all the parents of the corresponding entity has been created. To manage that, we have a TaskSubmitter class that handles the submissions of these CreationTasks to the ExecutorService, submitting a CreationTask for an entity only after all parents of the entity have been created. And for the actual execution of the entity creation, we have an EntityCreator class to process the model represented by the CreationTask, create the model on the cloud, then notify the TaskSubmitter that the model is completed so that the TaskSubmitter can submit CreationTasks for the model's children as appropriate.

We decided that it is the best if the TaskSubmitter class can check for completed models at its own convenience, therefore we designed the notification system to use a BlockingQueue. The EntityCreator class submits an object to the BlockingQueue after that object has been successfully created on the cloud. The TaskSubmitter class continually checks for completed models, blocking to wait for new models to be placed on the BlockingQueue. Whenever a new model is added to the BlockingQueue, the TaskSubmitter class removes the object from the BlockingQueue, then checks to see whether its children are ready to be processed, and if so submit a task to the ExecutorService to create the child model too.

The ExecutorService can schedule the execution of the CreationTasks at its own pace. When invoked, the CreationTasks basically call into the EntityCreator to create the appropriate entities on the cloud.

4.3.2 Implementation

This section gives a detailed walkthrough of the different steps in an entity creation process. In the duration of our discussion, the current CloudEntityModel that is under the process of creation is called $\texttt{model}_{\texttt{cur}}$. Its parents are called $\texttt{models}_{\texttt{parent}}$, and its children are called $\texttt{models}_{\texttt{child}}$.

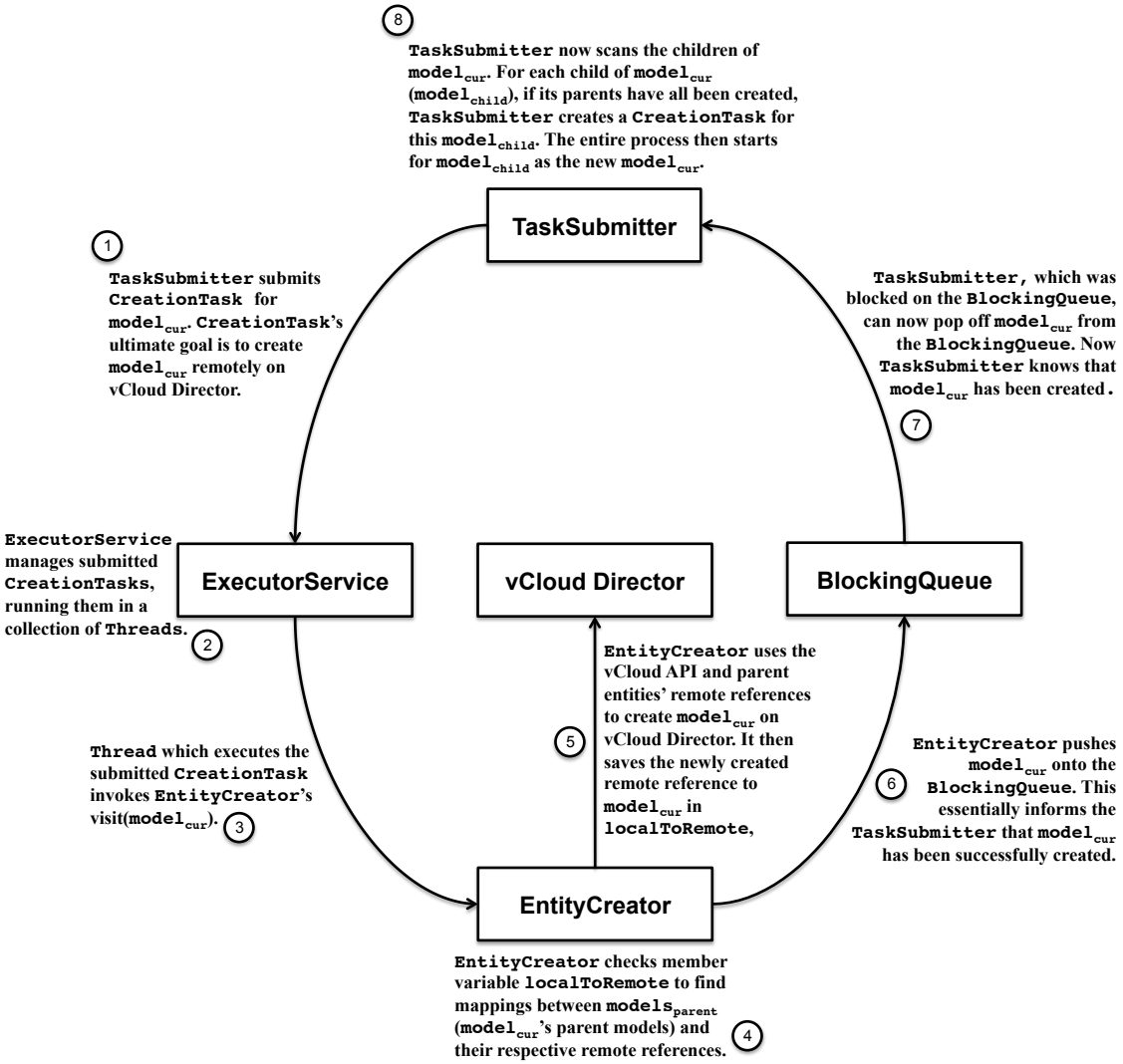

Figure 4-2: This diagram displays one cycle of the entity creation process. There is a total of 8 steps, and each steps are discussed in details in this section.

Step 1: `TaskSubmitter` starts the creation process

If a `CloudEntityModel`'s parents have all been created, the `TaskSubmitter` can start the creation process for it by creating a `CreationTask` for it. The `CloudEntityModel` in this case is named $\texttt{model}_{\texttt{cur}}$.

The `CreationTask` is a class that implements the `Callable` interface from the Java standard libraries. Note that even though the `CreationTask` has been created, it is not yet invoked. Instead, `TaskSubmitter` submits this `CreationTask` to the `ExecutorService`, which will eventually invoke this task when there are available `Thread`s.

In creating this `CreationTask`, `TaskSubmitter` provides it with references to both $\texttt{model}_{\texttt{cur}}$ and the `EntityCreator`. These references will be used later when this `CreationTask` is actually invoked.

Step 2: `ExecutorService` finds a free `Thread` for `CreationTask`

As discussed, the `ExecutorService` is an interface from the Java standard libraries that manages and executes any tasks that are submitted in a `Runnable` or `Callable` form. Our `CreationTask`s implements the `Callable` interface. We can use any implementation of `ExecutorService`. Currently we use the `ThreadPoolExecutor` implementation from the Java standard libraries so that we can parallelize our program with a pool of `Thread`s if we wish.

When the `ExecutorService` finally finds a free `Thread` for our `CreationTask`, it then invokes the `CreationTask`'s `call()` method.

Step 3: CreationTask invokes EntityCreator

As previously mentioned in our explanation of Step 1, our `CreationTask` has references to both $\texttt{model}_{\texttt{cur}}$ and the `EntityCreator`. The `CreationTask`'s ultimate goal is to create $\texttt{model}_{\texttt{cur}}$, which is a job for the `EntityCreator`.

`CreationTask` invokes $\texttt{model}_{\texttt{cur}}$'s `accept(CloudEntityVisitor)` method. Keep in mind that $\texttt{model}_{\texttt{cur}}$ is actually a `CloudEntityModel`, and therefore must have

implemented the `accept(CloudEntityVisitor)` method. All of the different implementations of `CloudEntityModel` have the same implementation of the method: `CloudEntityVisitor.visit(this)`. This ensures that the correct version of the `CloudEntityVisitor`'s `visit()` method is invoked. This is a use of the `Visitor Pattern` as previously discussed.

The `EntityCreator` class implements the `CloudEntityVisitor` interface, providing `visit()` methods for each implementation of the `CloudEntityModel`. As such, the `CreationTask` is basically invoking $\texttt{model}_{\texttt{cur}}\texttt{.accept(EntityCreator)}$, which in turns invoke $\texttt{EntityCreator.visit(model}_{\texttt{cur}}\texttt{)}$. This extra level of indirection added by the Visitor Pattern ensures that if $\texttt{model}_{\texttt{cur}}$ is a `CatalogModel`, for example, we are ultimately invoking `EntityCreator.visit(CatalogModel)`. This is important, since our `CreationTask` only knows that $\texttt{model}_{\texttt{cur}}$ is a `CloudEntityModel`, and does not know its specific implementation type.

Step 4: EntityCreator finds remote references for parent entities

No entity can be created on vCloud Director unless all of its parents are, because all creation calls to vCloud Director involves referencing these parent entities. The `EntityCreator` expects all $\texttt{models}_{\texttt{parent}}$ to be already created when a `visit()` method is invoked. This is not a problem in a single-threaded environment, since we can just determine one order to invoke all the `visit()` methods in. However, in a multi-threaded environment, one must take care to ensure that the `visit()` methods are invoked in the correct order. Our approach guarantees this because only `CloudEntityModel`s whose parents have all been created are given a `CreationTask`, and `EntityCreator`'s `visit()` methods can only be invoked via `CreationTasks`.

Even if all the parent entities of $\texttt{model}_{\texttt{cur}}$ have been created already, it turns out that the `CloudEntityModel` representation of these parent entities ($\texttt{models}_{\texttt{parent}}$) cannot be directly used in our creation calls to vCloud Director. We must use the reference types that vCloud Director expects, known as the **remote references**. This remote reference can be different kinds of objects depending on the implementation. For our implementation, we chose to communicate with vCloud Director via an in-

ternal Java SDK instead of making raw HTTP calls, therefore the remote references in this case are simply the interal SDK's object representation of the remote entities. If we had chosen to make raw HTTP calls directly using the vCloud API, then the remote reference in that case might be the HREF of the remote entity.

To ensure that when we are creating any sorts of entities that we will still have the correct corresponding remote references of the entities' parents, the `EntityCreator` keeps a local `Map`, known as `localToRemote`, that represents a mapping from our `CloudEntityModel`s to their remote references.

Step 5: EntityCreator creates entity on vCloud Director

Now that the `EntityCreator` knows the remote references of $\texttt{models}_{\texttt{parent}}$, it can use this information, along with the state of $\texttt{model}_{\texttt{cur}}$, to create $\texttt{model}_{\texttt{cur}}$ remotely on vCloud Director using the vCloud API. `EntityCreator` now waits until the creation has been completed remotely, then adds $\texttt{model}_{\texttt{cur}}$ and the remote reference of $\texttt{model}_{\texttt{cur}}$ into `localToRemote`.

Note that the `visit()` methods of `EntityCreator` can be simultaneously invoked by many different `Thread`s at once. The only local structure that may be written to at the same time by these different method invocations is the `localToRemote Map`. To preserve consistency, `localToRemote` is implemented by a `ConcurrentHashMap`. Although the `CloudEntityModel`s may be read at the same time, they are immutable: no write operations are performed on the `CloudEntityModel`s after the entire cloud entity model graph is created by the parser. The concurrency consistency of remote references are handled by vCloud Director.

Step 6: EntityCreator announces entity creation via BlockingQueue

The `EntityCreator` can now finally announce that $\texttt{model}_{\texttt{cur}}$ has been created. It does so by pushing $\texttt{model}_{\texttt{cur}}$ into the `BlockingQueue`. Only `CloudEntityModel`s that have already been created can be placed onto the `BlockingQueue`.

Listing 4.14 recaps all the steps that are taken by the `EntityCreator` in pseudocode.

```
Retrieve all models_parent
Retrieve remote references for all models_parent
Create model_cur on vCloud Director based on available info
Save remote reference to model_cur to localToRemote
Mark model_cur as completed by adding it to BlockingQueue
```

Listing 4.14: Pseudocode for `visit()` Method

Step 7: `TaskSubmitter` learns about entity creation via BlockingQueue

Meanwhile, the `TaskSubmitter`, after having submitted the `CreationTask`s for all `CloudEntityModel`s whose parents have all been created, is waiting to find out what new entities have been created. To do that, it simply keeps watch on the `BlockingQueue`. In fact, it calls `BlockingQueue`'s `take()` method, which is a method that blocks until an object has been placed on the `BlockingQueue`, before removing and returning that object.

After `EntityCreator` places $\texttt{model}_{\texttt{cur}}$ on the `BlockingQueue`, the `TaskSubmitter`, when free, will be able to take it off the `BlockingQueue`. The `TaskSubmitter` now knows that $\texttt{model}_{\texttt{cur}}$ has been created. It will keep a reference to $\texttt{model}_{\texttt{cur}}$ in `completedModels`, which is a member variable of the `TaskSubmitter` class that saves the `Set` of `CloudEntityModel`s that have been created so far.

Step 8: `TaskSubmitter` determines next entities to be created

Now that the `TaskSubmitter` knows that $\texttt{model}_{\texttt{cur}}$ has been created, it scans through $\texttt{model}_{\texttt{cur}}$'s children ($\texttt{models}_{\texttt{child}}$). For every $\texttt{model}_{\texttt{child}}$, `TaskSubmitter` must determine its current status. There are three possible statuses for all `CloudEntityModel`s. First of all, all `CloudEntityModel`s start off in the `PARENTS_NOT_CREATED` status. A `CloudEntityModel` will remain in this status until all of its parents have been created. After all its parents have been created, then the `CloudEntityModel` proceeds to the `PARENTS_CREATED` status. A `CloudEntityModel` remains in this status until it has been processed by the `EntityCreator` and remotely created on vCloud Director. After it has been created, it is now upgraded to the `CREATED` status.

Obviously, a $\texttt{model}_{\texttt{child}}$ only has two possible statuses: PARENTS_NOT_CREATED and PARENTS_CREATED. They cannot possibly be in the CREATED state, since we have already made it clear that no CreationTask can be submitted for a child until the TaskCreator acknowledges that its parents have been created. The TaskSubmitter must determine the $\texttt{models}_{\texttt{child}}$'s statuses by analysing them and their parents individually. For each $\texttt{model}_{\texttt{child}}$, the TaskSubmitter must examine its parents to see if they have all been created. If at least one parent has not been created yet, the TaskSubmitter will move on to the next $\texttt{model}_{\texttt{child}}$. If all parents have been created, then the TaskSubmitter knows that this particular $\texttt{model}_{\texttt{child}}$ is in the PARENTS_CREATED state, and thus can now be processed. The TaskSubmitter promptly creates a CreationTask for this $\texttt{model}_{\texttt{child}}$, and submits this CreationTask to the ExecutorService. This begins yet another creation process, and we are back to Step 1, with our $\texttt{model}_{\texttt{child}}$ as the new $\texttt{model}_{\texttt{cur}}$.

After the TaskSubmitter creates CreationTasks for each $\texttt{model}_{\texttt{child}}$ with the PARENTS_CREATED status and submits them to the ExecutorService, it can go back to waiting on the BlockingQueue for more completed CloudEntityModels.

The TaskSubmitter will keep looping in this process until all CloudEntityModels have reached the CREATED status. Because the TaskSubmitter is able to acknowledge all entity creation by taking created CloudEntityModels off the BlockingQueue, it can keep count of how many CloudEntityModels have been created so far, and how many are left to be created.

Listing 4.15 shows pseudocode for the TaskSubmitter's creation loop.

```
while there are CloudEntityModels not associated with a CreationTask:
    wait on BlockingQueue for a created CloudEntityModel
    for each child of completed CloudEntityModel:
        if all parents are completed:
            create and submit CreationTask to the ExecutorService
```

Listing 4.15: Pseudocode for TaskSubmitter

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Usage Comparison

The main goal of the vCloud Populator project is to conserve the time and resources spent on manual cloud population by providing an easy mean of cloud population automation. To provide a general idea of how much time and resources may be saved by automating the cloud population process, we have included a series of screenshots from a manual organization creation process through the web portal.

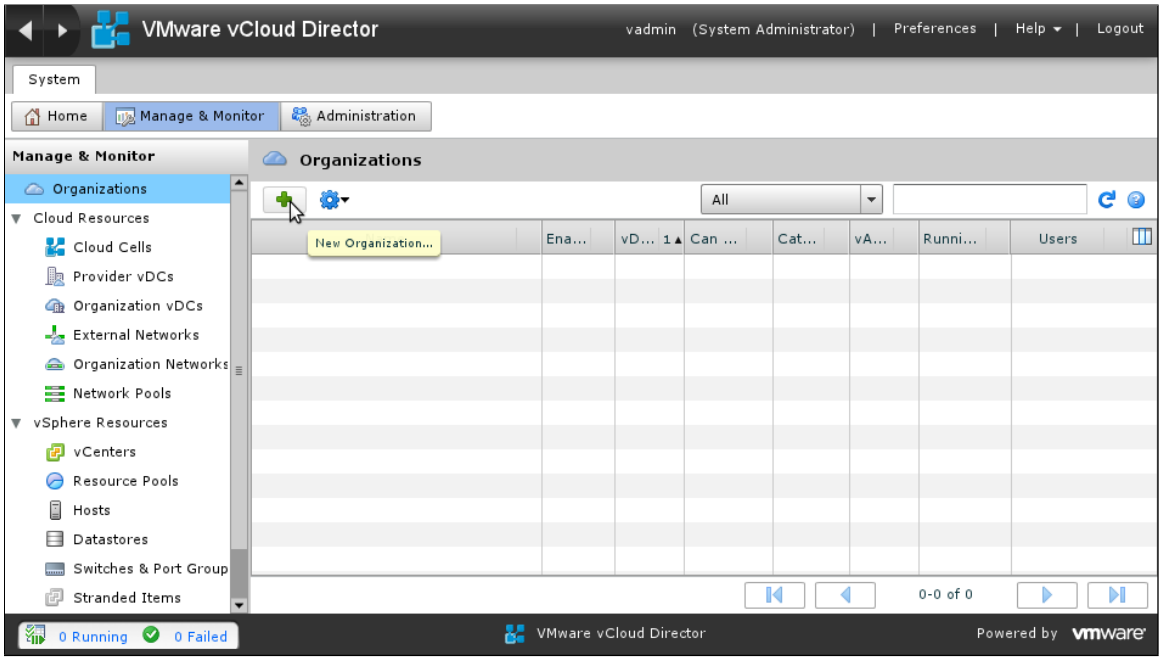

Figure 5-1: Click on the "+" sign under "Organizations" to add a new organization.

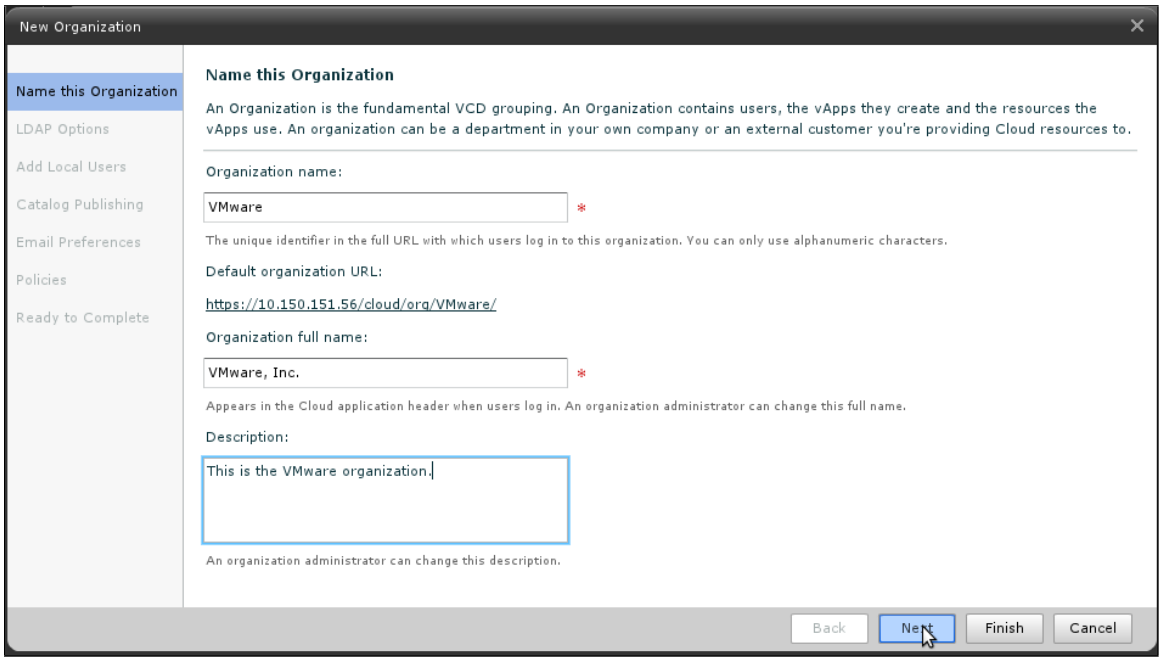

Figure 5-2: Fill in the name, full name, and description of the organization. Click "Next".

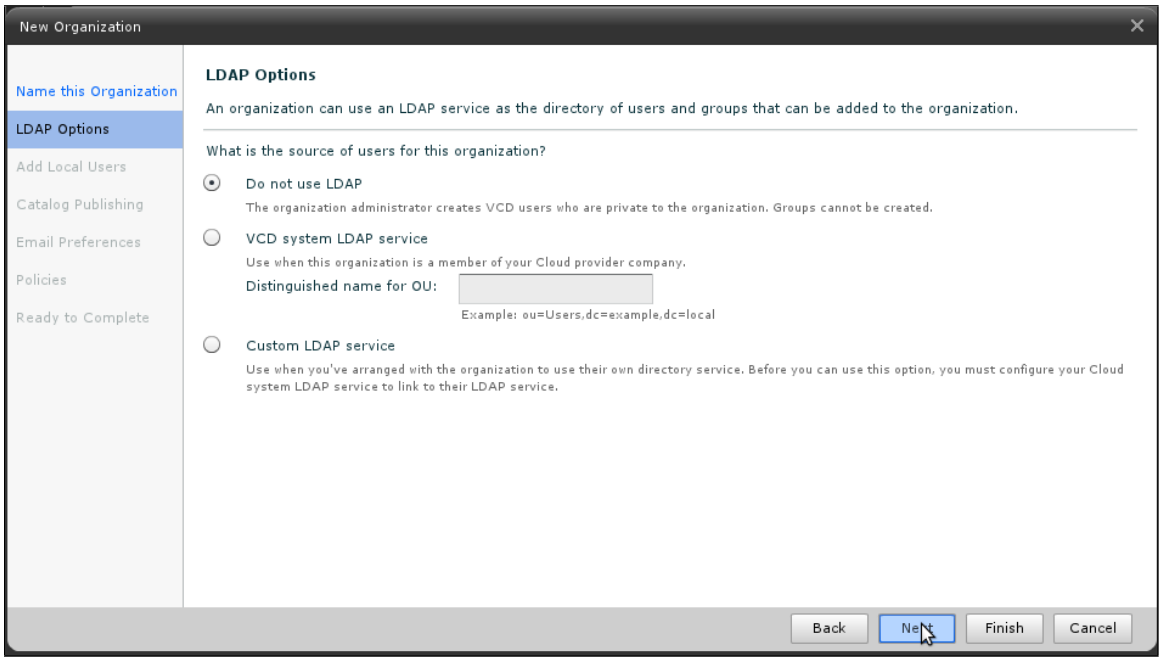

Figure 5-3: Choose between the available options for how users are managed. Note that the vCloud Populator does not support using LDAP for organizations right now.

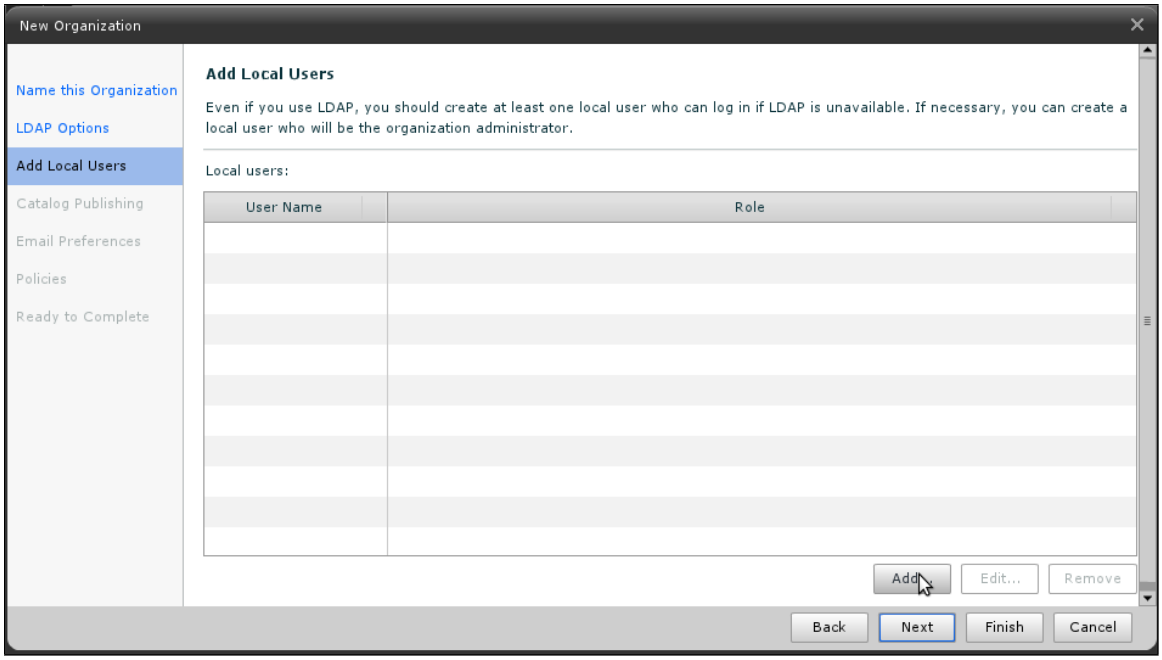

Figure 5-4: Here we can choose to add users directly. Click "Add" to add a user.

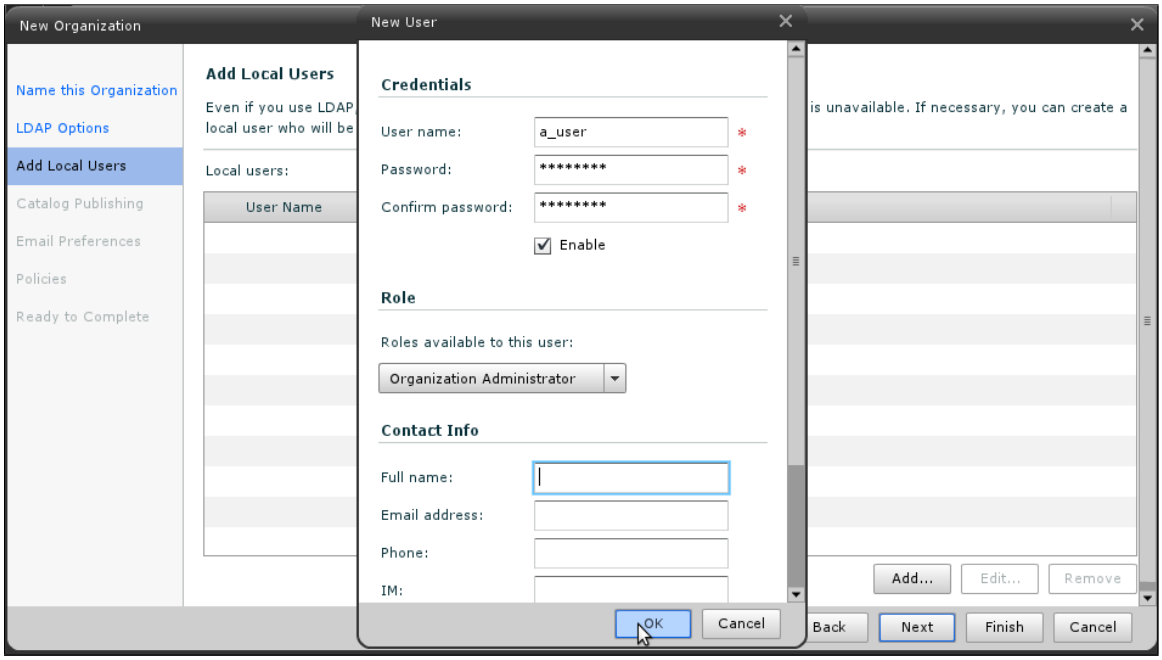

Figure 5-5: Fill in the information for this new user. Click "OK" when you are done.

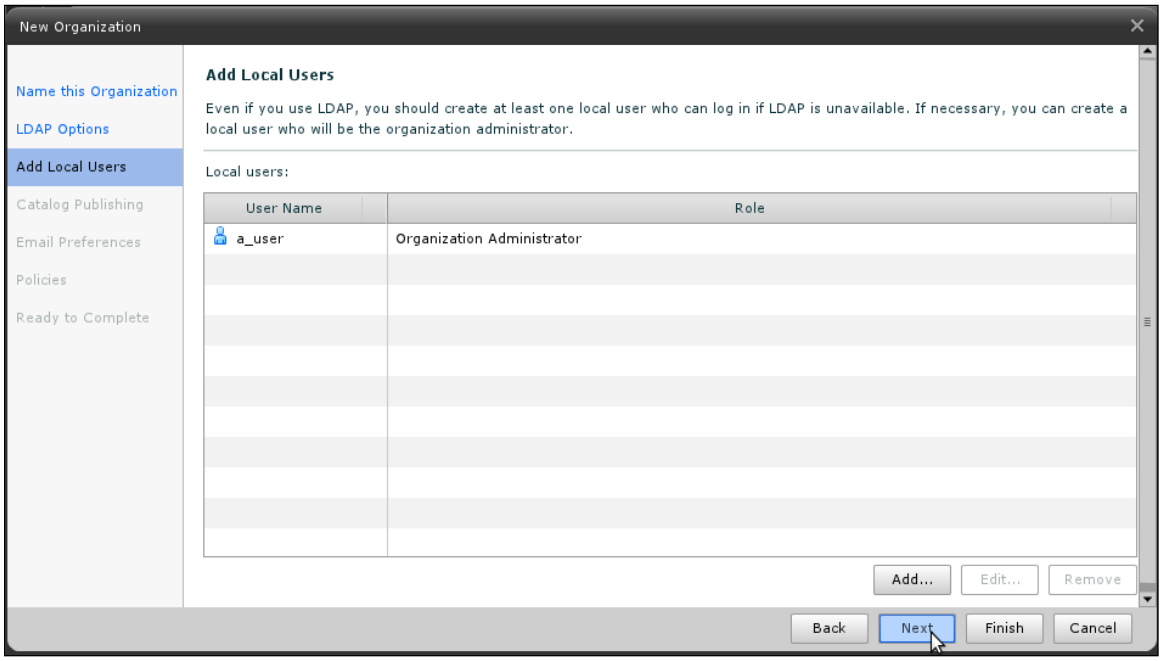

Figure 5-6: Keep adding users until you are done. Click "Next" to continue after adding all your users.

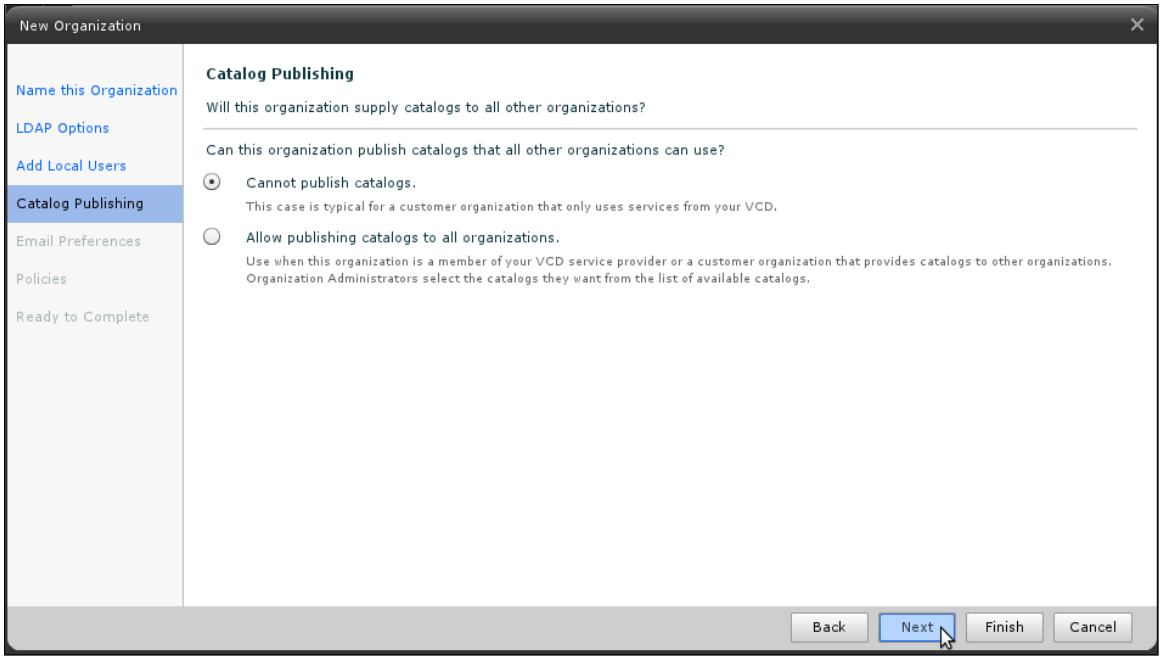

Figure 5-7: Choose whether or not your organization can publish catalogs. Click "Next" to continue.

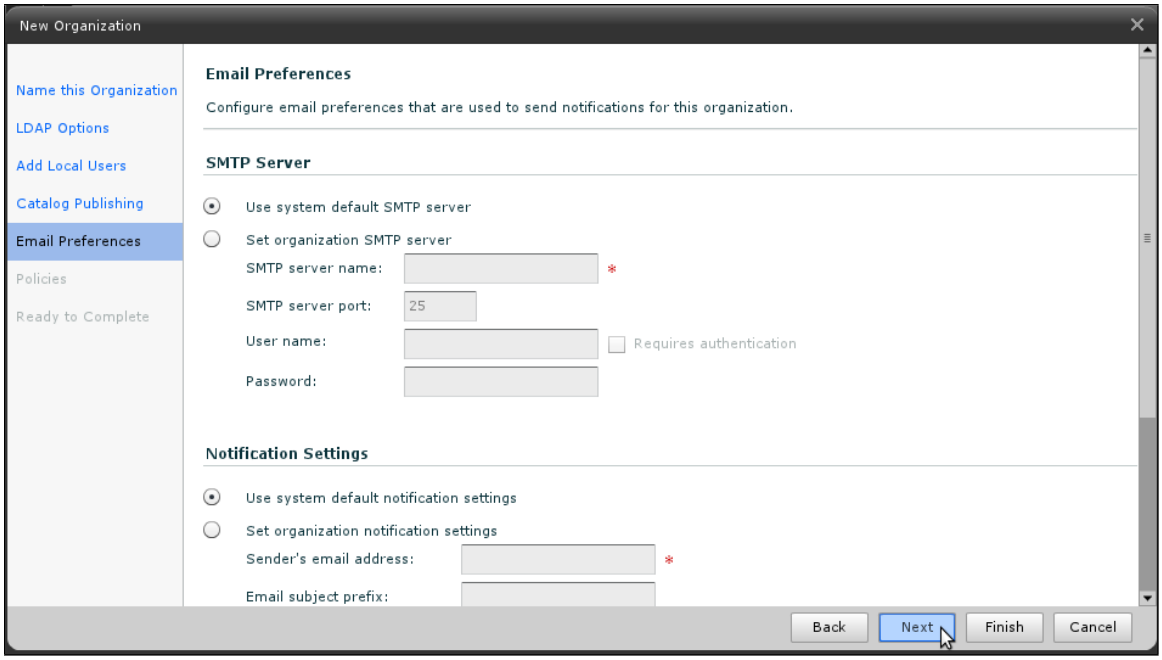

Figure 5-8: Fill in your email settings. Click "Next" to continue.

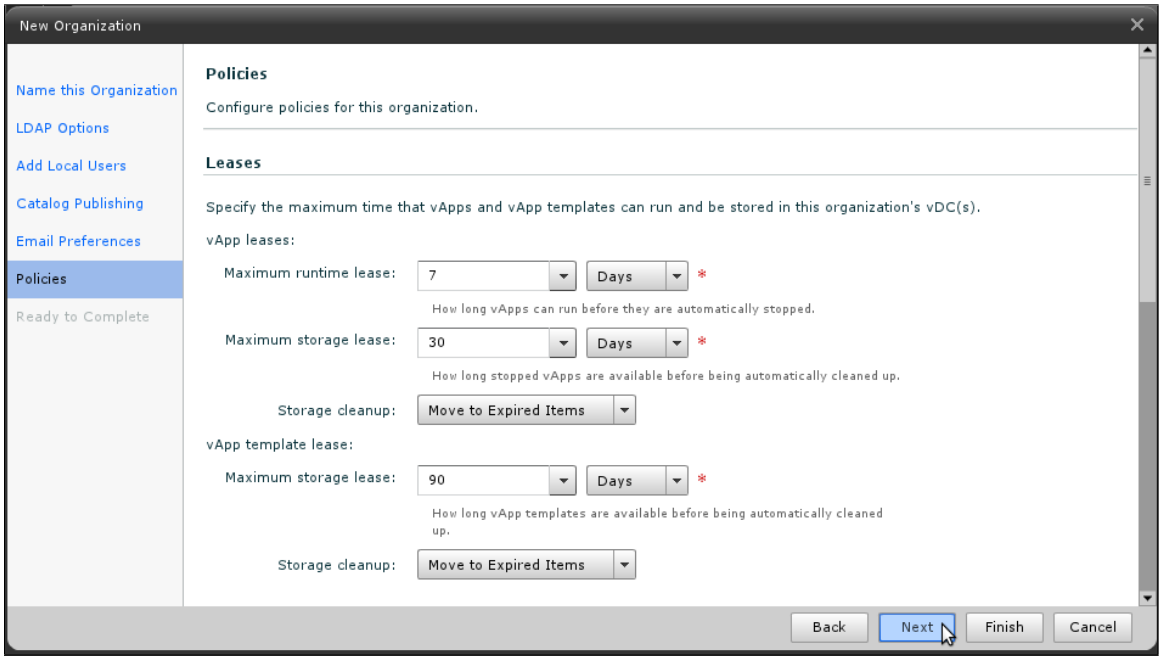

Figure 5-9: Fill in your policies settings. Click "Next" to continue.

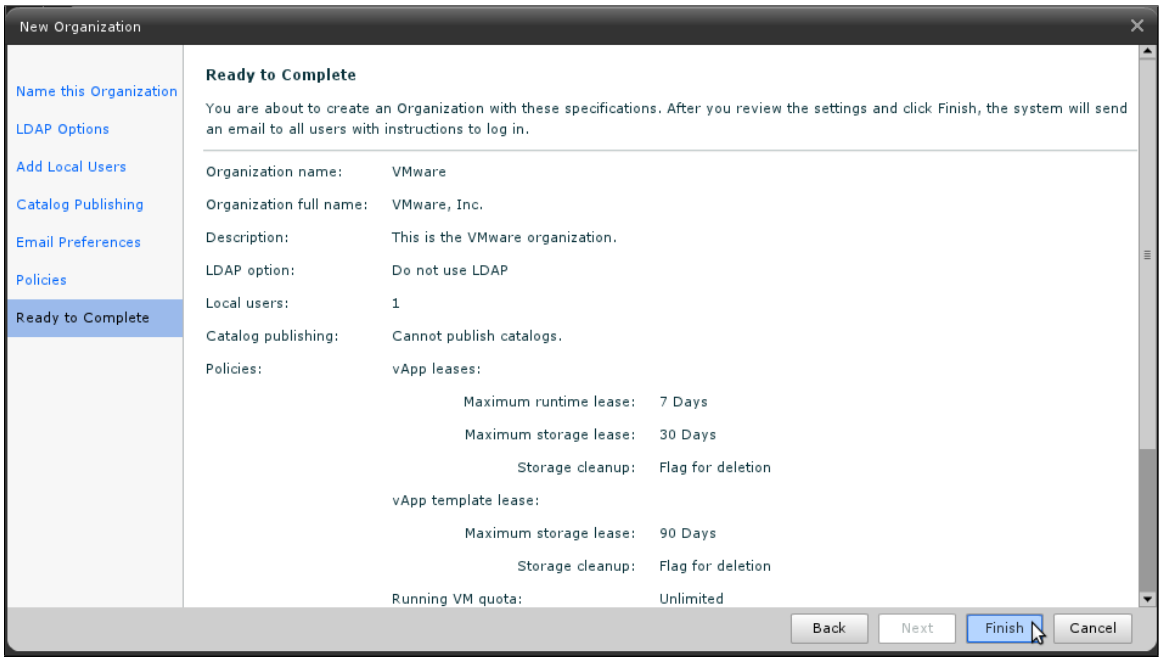

Figure 5-10: Review all your information, then click "Finish" to complete the organization creation.

Obviously, going through the user interface to manually create entities make sense in the normal scenario of a service provider creating a new organization. He must of course manually configure all settings to ensure that everything matches the contract between him and his client organization. However, for developers who, for example, just need an organization on the cloud so that they can test the media upload functionality, the availability of the numerous options to fill in is extraneous. Compare the previous steps (clicking through 10 screens) to create an organization with a user with the following vCSL script when we do not care about most of the details:

```
include "stdlib.vcsl"

org(name: "VMware", fullname: "VMware, Inc.",
    description: "This is the VMware organization.") {
    user(name: "a_user", password: "password", role: role_org_admin)
}
```

Listing 5.1: Sample vCSL Script to Create an Organization with a User

Writing the vCSL script will definitely be faster than manually creating the organization through the user interface. The vCSL script has the added advantage of possible reuse, whereas if a developer wishes to recreate an organization with the same settings as before, he must go through the entire manual process once more. And in situations where the developer must create many copies of the same entity, the numerical declaration syntax will definitely aid the developer in cutting down the time it takes to write the script, whereas manual creation through the user interface will become more and more tedious as the number of entities to create rises.

Take adding a user to an organization, for example. If there are 100 users to be added, then manual creation through the user interface involves repeating Step 4 and 5 a hundred times, whereas the vCSL script will only take several lines to write as long as information about the users to be created are already in an accessible CSV file, as shown in Listing 5.2.

Of course, the actual entity creation on the server still takes the same amount of time. However, since everything is automated, it is now possible for the developer

to start the cloud population process, then engage in other work until the entire process has completed. This allows the developer to focus on other aspects of their work for the entire duration of the automated cloud population, instead of needing to continuously check back on an entity's creation status to create children entities after the parent entity has been completed.

```
include "stdlib.vcsl"

org(name: "VMware", fullname: "VMware, Inc.",
    description: "This is the VMware organization.") {
    20.user(role: role_org_admin)
    20.user(role: role_catalog_author)
    20.user(role: role_console_access)
    20.user(role: role_vapp_author)
    20.user(role: role_vapp_user)
}
```

Listing 5.2: Sample vCSL Script to Create an Organization with 100 Users

Chapter 6

Conclusion and Future Work

This thesis presents a new approach to automating the vCloud Director population process, in hopes of conserving the valuable time and resources of developers who presently must complete the process manually. The major contributions of this thesis included a domain-specific language named vCSL and a programmatic system to populate a vCloud Director instance based on an input vCSL script.

There are still many different areas that will benefit from further work. Some ideas are direct improvements on the vCloud Populator, while others are major projects on their own that can make great use of the vCloud Populator. We list some of these ideas below.

6.1 Unsupported Entities and Properties

First and foremost, the vCloud Populator project at its current state does not handle certain entities and properties. For example, networking is currently not supported in vCloud Populator. LDAP support for organizations also has not been implemented yet, so it is not possible for an end user to create an organization configured to use LDAP. Extending support for these functionalities plays a major role in any future work.

6.2 Cloud Entity Model Graph Integrity Checker

vCloud Populator's execution engine currently expects the input cloud entity model graph to be acyclic. No parent entity should ever be a direct descendant of its children entities, otherwise no creation order can satisfy the population problem. This may sound obvious, but note that some entities in VMware vCloud Director may cause cycles in the graph. For example, a vApp may have a vApp template as its parent, since a vApp can be instantiated from a vApp template. However, a vApp template may also have a vApp as a parent, since a vApp template can be created from a vApp through a capture operation. Fortunately, the design of the vCSL enforces acyclicity. All parents are referred to either through the use of the hierarchical declaration or through the use of variables, and therefore the order of entity declaration prevents an entity that is declared later to be the parent of an entity that is declared earlier. Thus it is not possible to create a cyclic cloud entity model graph from a vCSL script. However, if the input cloud entity model graph has been created manually or through other automated means, it is certainly possible to create a cycle in the parent-child relationships. We hope to incorporate a cloud entity model graph integrity checker in our future work.

6.3 Cloud Verification

Sometimes it is necessary to compare two cloud states to see if they are equivalent. Although this use case is not directly related to cloud population, it is expected that a cloud verification execution engine can be built upon the vCSL parser and cloud models, similar to how the vCloud Populator execution engine was implemented. This project will have two main components: an extractor, which uses the vCloud API to communicate with a vCloud Director instance and construct a cloud entity model graph that represents the state of the instance, and a verifier, which compares two cloud entity model graph to check if they are equivalent. With these two components along with the relevant components in the vCloud Populator, the cloud verification

execution engine can then compare a user-written vCSL script with an actual cloud instance to verify that the two cloud states are essentially the same.

A major challenge of the cloud verification project lies in the definition for equivalence. For example, two organizations that are both created from the `org()` vCSL declaration with no enumerated properties probably should be considered equivalent, even though they may have different generated names in the cloud entity model graph. To do that, we must distinguish values that are generated by either the basic default mechanism or the `data` keyword from actual data that the end user supplied through the named arguments. Another challenge is to determine the actual graph comparison algorithm to be used for the cloud entity model graphs.

This project has many potential use cases, and we hope to be able to make use of the vCloud Populator for this project in the future.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

EBNF for vCSL

This appendix shows the Extended BackusNaur Form (EBNF) [21] of vCSL.

```
(* A vCSL script consists of zero or more declarations. *)
vCSLScript = {declaration} ;

(* A declaration can be either an entity declaration
   or a functional declaration. *)
declaration = entityDecl | funcDecl ;

(* A functional declaration is either a common declaration,
   a data declaration, or an include declaration. *)
funcDecl = commonDecl | dataDecl | inclDecl ;

(* An include declaration consists of the include keyword
   followed by the file path of the include file. *)
inclDecl = includeKeyword , filePath ;

(* A data declaration consists of the data keyword
   followed by the file path of the data file. *)
dataDecl = dataKeyword , filePath ;

(* A common declaration consists of the common keyword,
   its arguments, followed by an optional closure. *)
commonDecl = commonKeyword , arguments , [closure] ;

(* A numerical declaration may precede an entity declaration. *)
entityDecl = [number , "."] , oneEntityDecl ;
```

```
(* A single entity declaration consists of the entity keyword,
   its arguments, followed by an optional closure. *)
oneEntityDecl = entityKeyword, arguments , [closure] ;

(* Arguments are properties surrounded by parentheses. *)
arguments = "(" , properties , ")" ;

(* Each property is separated by commas *)
properties = [prop , {"," , prop}] ;

(* A property is a mapping from a keyword to a value. *)
prop = propKeyword , ":" , propValue ;

(* A property value can either be a simple value (Integer, String)
   or a complex value (for a complex attribute). *)
propValue = simplePropValue | complexPropValue ;

(* A complex property value is represented as a list of
   sub-properties contained within square brackets. *)
complexPropValue = "[" , properties , "]" ;

(* A closure is a block of declarations enclosed in curly braces. *)
closure = "{" , {closureDecl} , "}" ;

(* Allowed declarations in a closure are the entity declarations
   and the common declarations. *)
closureDecl = entityDecl | commonDecl ;
```

Table A.1: vCSL EBNF Terminals

includeKeyword	the `include` keyword
dataKeyword	the `data` keyword
commonKeyword	the `common` keyword
entityKeyword	an entity keyword, such as `org` or `vapp`
propKeyword	a property keyword, such as `name` or `description`
simplePropValue	primitives, such as an `Integer` or a `String`
filePath	a `String` representing the path to a file
number	an `Integer`

Appendix B

Configuration State Specification

Below is a number of tables showing the different entities and their properties in the vCSL. Bolded properties represent complex properties. Italicized properties represent required properties. Properties followed by a * represent properties that are not filled in by the basic default mechanism.

B.1 Supported Entities

Table B.1: This table shows information about all the vCloud Director entities that are supported by the vCloud Populator. It shows the corresponding entity keyword for each entity type, along with each entity type's parents.

entity type	**keyword**	**owner**	**other parents**
organization	org		
provider vDC	pvdc		
role	role		
org vDC	orgvdc	organization	provider vDC
catalog	catalog	organization	
user	user	organization	role
media	media	org vDC	catalog
vApp template	vapptemplate	org vDC	catalog, vApp
vApp	vapp	org vDC	vApp template

B.2 Provider vDC Properties

Table B.2: Supported Attributes for `pvdc` Keyword

*name**	Name of the existing provider vDC.

B.3 Role Properties

Table B.3: Supported Attributes for `role` Keyword

*name**	Name of the existing role.

B.4 Organization Properties

Table B.4: Supported Attributes for `org` Keyword

name	Name of the organization (may not contain spaces).
fullName	Full name of the organization (may contain spaces).
description	Description of the organization.
enabled	Boolean to indicate whether this organization is enabled.
settings	Settings of this organization.

Table B.5: Supported Attributes for `org:settings` Complex Attribute

emailSettings	Email settings of the organization.
vAppLeaseSettings	Settings about this organization's vApp leases.
vAppTemplateLeaseSettings	Settings about this organization's vApp template leases.
passwordPolicySettings	Password policy settings of this organization.
generalSettings	General settings of the organization.

Table B.6: Supported Attributes for `org:settings:passwordPolicySettings` Complex Attribute

accountLockoutEnabled	Boolean to indicate whether the account lockout mechanism is enabled.
accountLockoutInterval	Number of minutes before a locked account is accessible again.
invalidLoginsBeforeLockout	Number of invalid login attempts that will trigger account lockout.

Table B.7: Supported Attributes for `org:settings:vAppLeaseSettings` Complex Attribute

deleteOnStorageLeaseExpiration	Boolean to indicate whether a vApp is by default deleted immediately upon storage lease expiration. If false, the storage is flagged for deletion but not immediately deleted.
deploymentLease	Default duration of a vApp deployment lease in seconds.
storageLease	Default duration of a vApp storage lease in seconds.

Table B.8: Supported Attributes for `org:settings:vAppTemplateLeaseSettings` Complex Attribute

deleteOnStorageLeaseExpiration	Boolean to indicate whether a vApp template is by default deleted immediately upon storage lease expiration. If false, the storage is flagged for deletion but not immediately deleted.
storageLease	Default duration of a vApp template storage lease in seconds.

Table B.9: Supported Attributes for `org:settings:generalSettings` Complex Attribute

canPublishCatalogs	Boolean to indicate whether this organization is allowed to publish catalogs to users from other organizations.
delayAfterPowerOn	Default number of seconds to delay for a virtual machine boot after it is powered on.
deployedVmQuota	Maximum number of virtual machines that can be deployed simultaneously by a member of this organization.
storedVmQuota	Maximum number of virtual machines that can be stored by a member of this organization.
useServerBootSequence	Boolean to indicate whether virtual machines in this organization uses the server's boot sequence by default.

Table B.10: Supported Attributes for the `org:settings:emailSettings` Complex Attribute

defaultSubjectPrefix	Default prefix to use in the subject line of system email notifications.
fromEmailAddress	Email address from which to send system email notifications.
alertEmailToAllAdmins	Boolean to indicate whether system email notifications should be sent to all users with the Organization Administrator role.
defaultOrgEmail	Boolean to indicate whether this organization uses the system's default email properties.
defaultSmtpServer	Boolean to indicate whether this organization uses the system's default SMTP server.
smtpServerSettings	Configuration settings for the organization's SMTP server.

Table B.11: Supported Attributes for `org:settings:emailSettings:smtpServerSettings` Complex Attribute

host	Host name of the SMTP server.
useAuthentication	Boolean to indicate whether the SMTP server uses authentication.
username*	Username to use when logging into the SMTP server. Required if useAuthentication is true.
password*	Password to use when logging into the SMTP server. Required if useAuthentication is true.

B.5 Org vDC Attributes

Table B.12: Supported Attributes for `orgvdc` Keyword

name	Name of this org vDC.
description	Description of this org vDC.
allocationModel	The allocation model in use by this org vDC. The valid values are AllocationVApp, AllocationPool, and ReservationPool.
enabled	Boolean to indicate whether this org vDC is enabled.
cpuCapacity	The CPU capacity that is available to this org vDC.
memoryCapacity	The memory capacity that is available to this org vDC.
storageCapacity	The storage capacity that is available to this org vDC.
resourceGuaranteedCpu	Number between 0.0 to 1.0 representing the percentage of allocated CPU resources that are guaranteed to deployed vApps in this org vDC.
resourceGuaranteedMemory	Number between 0.0 to 1.0 representing the percentage of allocated memory resources that are guaranteed to deployed vApps in this org vDc.
vmQuota	Maximum number of virtual machines that can be allocated in this org vDC. Enter 0 for unlimited.
thinProvisioning	Boolean to indicate whether virtual machines allocated in this org vDC uses thin provisioning, which allocates storage to virtual machines only on demand rather than at creation time.
fastProvisioning	Boolean to indicate whether virtual machines allocated in this org vDC uses fast provisioning, which may reduce the time it takes to create virtual machines by using vSphere linked clones.
vcpu	The clock frequency, in MHz, for each virtual CPU core that is provisioned in this org vDC.
*org**	The organization that this org vDC belongs to.
*pvdc**	The provider vDC backing this org vDC.

Table B.13: Supported Attributes for `orgvdc:cpuCapacity`, `orgvdc:memoryCapacity`, and `orgvdc:storageCapacity` Complex Attributes

units	The units in which this resource capacity is specified. For example, MB for memory resource or MHz for CPU resource.
limit	The maximum amount of resource that can be allocated, specified in the given units.
allocated	The quantity of the resource to be allocated.

B.6 User Attributes

Table B.14: Supported Attributes for `user` Keyword

name	Username of the user (may not contain spaces).
description	Description of the user.
alertEmailAddress	Email address to which alert emails for this user should be sent.
alertEmailPrefix	Prefix string for alert emails that are sent to this user.
deployedVmQuota	Number of VMs that this user can have simultaneously deployed. Enter 0 for unlimited.
emailAddress	Regular email address for this user.
fullName	Full name of the user.
im	Instant messenger address of the user.
alertEnabled	Boolean to indicate whether alerts are enabled for this user.
defaultCached	Boolean to indicate whether this user is cached.
enabled	Boolean to indicate whether this user is enabled.
password	Password of the user.
storedVmQuota	Number of VMs that this user can have simultaneously stored. Enter 0 for unlimited.
telephone	Telephone number of this user.
*org**	The organization that this user belongs to.
*role**	The role that this user has.

B.7 Catalog Attributes

Table B.15: Supported Attributes for `catalog` Keyword

name	Name of the catalog.
description	Description of the catalog.
*org**	The organization that this catalog belongs to.

B.8 VApp Attributes

Table B.16: Supported Attributes for `vapp` Keyword

name	Name of the vApp.
deploy	Boolean to indicate whether the vApp should be deployed after a successful instantiation.
powerOn	Boolean to indicate whether the vApp should be powered on after a successful instantiation.
description	Description of the vApp.
allEulasAccepted	Boolean to indicate whether all EULAs associated with the vApp template is accepted.
deleteSource	Boolean to indicate whether the original vApp template should be deleted after a successful instantiation. (only if vApp template is provided)
vapptemplate*	The vApp template that this vApp should be instantiated from (if not provided, will create empty vApp).
*orgvdc**	The org vDC that this vApp should be created in.

B.9 VApp Template Attributes

Table B.17: Supported Attributes for `vapptemplate` Keyword

name	Name of the vApp template.
description	Description of the vApp template.
*file**	The OVF descriptor file that this vApp template should be created from.
catalog*	The catalog that this vApp should be placed into.
*orgvdc**	The org vDC that this vApp template should be stored in.

B.10 Media Attributes

Table B.18: Supported Attributes for `media` Keyword

imageType	The type of media image. Valid values are ISO and FLOPPY.
name	Name of the media.
description	Description of the media.
*file**	The file to be uploaded for this media.
catalog*	The catalog that this vApp should be placed into.
*orgvdc**	The org vDC that this media should be stored in.

Appendix C

Vocabularies

attribute	Information and settings of an entity. For example, a entity's name is an attribute.
basic default mechanism	A mechanism in the vCloud Populator that provides default values for most attributes of an entity.
child	An entity that depends on another entity.
cloud entity	An abstraction introduced in vCloud Director, such as an organization, a user, or a vApp.
configuration	All information about an entity, besides its children.
entity keyword	A keyword in the vCSL that represents a cloud entity.
named arguments	A list of arguments where each argument can be referred to by argument name.
owner	A special type of parent who owns another entity. An owner is also a parent.
parent	An entity that the current entity depends on.
property	An attribute or a parent of an entity. For example, an org vDC's and its owner organization are both properties of the org vDC.
property keyword	The name of a property.
state (entity)	Entire bundle of information about an entity, which includes both its configuration and its children.
state (cloud)	Entire bundle of information about a cloud, this includes all of the cloud entity and their states.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix D

stdlib.vcsl

Because vCloud Director has a series of predefined entities, we have compiled a standard library file that can be included in every file to save the users time from declaring these predefined entities. This file is called the `stdlib.vcsl`. Currently it includes all the predefined roles in vCloud Director. As more predefined entities are established, they will be added to this file in future updates.

```
role_catalog_author = role(name:"Catalog Author")
role_console_access = role(name:"Console Access Only")
role_org_admin = role(name:"Organization Administrator")
role_sys_admin = role(name:"System Administrator")
role_vapp_author = role(name:"vApp Author")
role_vapp_user = role(name:"vApp User")
```

Listing D.1: `stdlib.vcsl`

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] VMware, "VMware vCloud Director: Secure Private Clouds, Infrastructure as a Service." http://www.vmware.com/products/vcloud-director/overview.html. Accessed Dec. 18, 2011.

[2] VMware, "VMware vSphere: Private Cloud Computing for Mid-Size & Enterprise Businesses." http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/overview.html. Accessed Dec. 18, 2011.

[3] VMware, "VMware Communities: VMware vCloud API." http://communities.vmware.com/community/vmtn/developer/forums/vcloudapi. Accessed Dec. 18, 2011.

[4] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[5] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Lea, "NIST Cloud Computing Reference Architecture," tech. rep., National Institute of Standards and Technology, 2011.

[6] VMware, "VMware vSphere Basics." http://pubs.vmware.com/vsphere-50/topic/com.vmware.ICbase/PDF/vsphere-esxi-vcenter-server-50-basics-guide.pdf. Accessed Dec. 18, 2011.

[7] VMware, "vCloud Director Administrator's Guide." http://www.vmware.com/pdf/vcd_15_admin_guide.pdf. Accessed Dec. 18, 2011.

[8] VMware, "vCloud Director User's Guide." http://www.vmware.com/pdf/vcd_15_users_guide.pdf. Accessed Dec. 18, 2011.

[9] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[10] Codehaus Groovy Community, "Groovy - Home." http://groovy.codehaus.org. Accessed Dec. 18, 2011.

[11] F. Dearle, *Groovy for Domain-Specific Languages*. Packt Publishing, 2010.

[12] Codehaus Groovy Community, "Groovy - Closures - Formal Definition." http://groovy.codehaus.org/Closures+-+Formal+Definition. Accessed Dec. 18, 2011.

[13] Codehaus Groovy Community, "Extended Guide to Method Signatures." http://groovy.codehaus.org/Extended+Guide+to+Method+Signatures. Accessed Dec. 18, 2011.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, pp. 331–344. Addison-Wesley, fifth ed., 1995.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley, 2005.

[16] Oracle, "Executor (Java Platform SE 6)." http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Executor.html. Accessed Dec. 18, 2011.

[17] Oracle, "Runnable (Java Platform SE 6)." http://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html. Accessed Dec. 18, 2011.

[18] Oracle, "Callable (Java Platform SE 6)." http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Callable.html. Accessed Dec. 18, 2011.

[19] Oracle, "ExecutorService (Java Platform SE 6)." http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html. Accessed Dec. 18, 2011.

[20] Oracle, "ThreadPoolExecutor (Java Platform SE 6)." http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ThreadPoolExecutor.html. Accessed Dec. 18, 2011.

[21] International Organization for Standardization, "Information technology - Syntactic metalanguage - Extended BNF," tech. rep., International Organization for Standardization, 1996.